8088
IBM PC
Assembly
Language
Programming

Gary Shade

8088/IBM PC Assembly Language Programming

Gary A. Shade

CBS Computer Books

HOLT, RINEHART AND WINSTON
New York Chicago San Francisco Philadelphia
Montreal Toronto London Sydney Tokyo
Mexico City Rio deJaneiro Madrid

Copyright© 1985 Gary A. Shade
All rights reserved.
Address correspondence to:
383 Madison Avenue, New York, NY 10017

First distributed to the trade in 1985 by Holt, Rinehart and Winston
general book division.

Library of Congress Cataloging in Publication Data

Shade, Gary A.
8088/IBM PC Assembly Language Programming

(CBS computer books)
Bibliography: p.
Includes index.
1. IBM Personal Computer—Programming. 2. Intel 8088 (Microprocessor)—Programming. 3. Assembler language (Computer programming language) I. Title. II. Series.
QA76.8.I2594S45 1985 005.2'65 85-14156
ISBN 0-03-001298-8

Printed in the United States of America

Published simultaneously in Canada

CBS COLLEGE PUBLISHING
Holt, Rinehart and Winston
The Dryden Press
Saunders College Publishing

Dedication

This book is dedicated to my son Jason. May you always communicate with authority and intelligence, and touch those and the world around you with your compassion and understanding.

Dad

Acknowledgements

CP/M is a trademark of Digital Research Incorporated.
MS-DOS is a trademark of Microsoft Corporation.
Z-80 is a registered trademark of Zilog Corporation.
iAPX, INTEL are registered trademarks of Intel Corporation.
6800 and 68000 are registered trademarks of Motorola Inc.
WordStar is a registered trademark of MicroPro International.

This book and all programming examples were written using WordStar from MicroPro International.

A special thanks to Intel for their cooperation and in granting reprint permissions for the many illustrations, tables, and data sheets that appear in this book.

Table of Contents

Preface xi
Chapter Review xiv

Chapter 1 **Numbering Systems** 1
Numbering Systems 2
Character Encoding 11
Chapter 1 Review 13

Chapter 2 **Assembly Language Programming** 15
The Programmer 16
Language Differences 17
Computer Languages 18
Assemblers 21
The Editor 23
Symbolic Representation 23
Forward References 26
Cross Assemblers 27
Linkers 27
Creating a Source Program 28
Operand Field 31
Your First AL Program 32
The Files You Have Created 38
Chapter 2 Review 40

Chapter 3 **8088 Architecture** 43
Microprocessors and Microcomputers 43
Bus Architecture (Hardware) 51
PC Memory Map 55
The 8088 Microprocessor 55
Chapter 3 Review 69

Chapter 4 **8088 Instruction Set** 71
Instruction Groups 72
Data Transfer Instructions 76
Arithmetic Instructions 87
Before You Continue 102
Bit Manipulation Instructions 102
Control and Transfer Instructions 112
String Manipulation Instructions 124
Processor Control Instructions 130
Chapter 4 Review 134

Chapter 5 **Assembler Features** **135**
Assembler Format 136
Data Pseudo-ops 136
Symbolic Names 142
More Pseudo-ops 145
Conditional Pseudo-ops 155
Listing Pseudo-ops 158
Operators 162
Chapter 5 Review 168

Chapter 6 **Macros and MS-DOS** **169**
Macro Definitions 169
MS-DOS/PC-DOS and Macros 180
Example Program 185
Chapter 6 Review 188

Chapter 7 **Numbering Conversions: A Programming Example** **189**
Purpose 189
Conceptual Design 190
Segment Definitions 192
Program Logic Flow 198

Chapter 8 **Disk I/O Programming** **207**
The Diskette 209
MS-DOS Programming Tools 213
Methods of File Access 221
Methods of File Access Under MS-DOS 2.0 226
Absolute Disk Access 229
Programming Examples 231
Chapter 8 Review 247

Chapter 9 **The Basic Input/Output System** **249**
Printer I/O 252
Video I/O 253
Graphic Modes 256
Sound 263
Chapter 9 Review 265

Chapter 10 **Communications** **267**
Methods of Analog Data Transmission 276
A Closer Look at Duplex 278
Asynchronous versus Synchronous Communications 282
Baud Rate 285
Error Checking 285
Modems 288
The Program: COMM.ASM 292
MS-DOS RS-232C Functions 302
BIOS Communication Functions 303
A Communications Program:
COMM.ASM and DLOAD.ASM 305

In Conclusion 317
Chapter 10 Review 318

Appendix A 321

Appendix B 327

Appendix C 347

Answers to Chapter Reviews 347

Appendix D 353

Bibliography 473

Index 475

Preface

This book is intended to be an introduction to Assembly Language programming on the IBM PC, and to provide the Assembly Language programmer with a valuable reference for future use. Three subject areas are of interest when writing programs for the IBM (and compatible computers). These are:

1. the 8088 microprocessor, the brain of the IBM;
2. MS-DOS or PC-DOS, the operating systems for the compatibles and the IBM; and
3. general programming practices, which when applied to Assembly Language programming, make the exercise worthwhile.

You should know how to operate your computer. This book will not instruct you on where the on/off switch is located, nor will it tell you how to properly insert a diskette and efficiently use DOS commands, such as DIR or COPY. You should consult your operations manual if these commands do not sound familiar to you.

Why Assembly Language?

Assembly Language is capable of creating programs that execute faster and are more efficient in terms of memory usage than are most higher level languages, such as BASIC. Most people have heard about Assembly Language (AL) and Assembly Language Programming (ALP), but few may have already attempted to program their machines using AL.

Assembly Language is intimidating to the novice programmer, but then so are numerous other activities when attempted for the first time.

You'll find that the skills acquired from learning to program your IBM PC will enable you to quickly learn a different microprocessor's architecture and instruction set. Once you become really proficient at Assembly Language programming, learning a new microprocessor's architecture and the instructions necessary to program it will take about a month.

The current trend in applications development work is to use higher level languages, such as BASIC, FORTH, and Pascal, to reduce programming time. However, most applications require some portion of the application program to be written in Assembly Language (AL).

High level languages such as BASIC and COBOL do speed up development time, but they do not execute as quickly as programs written in AL. Higher level languages also require more memory than programs written in Assembly Language.

Getting to Know Your Micro

Perhaps the most important concept any book on Assembly Language Programming (ALP) must convey is that you, the programmer, will become acquainted with the hardware that makes a computer a computer. This knowledge is essential when writing in Assembly Language. You must know the microprocessor's architecture, its registers and instructions.

As this book is about 8088/IBM ALP programming, you should have access to an IBM PC or one of the compatible computers with a minimum of 64K memory, two disk drives, PC-DOS or MS-DOS 2.0, the IBM Macro Assembler (IBM part number 60242002), and, optionally, a full screen editor or word processing software package such as WordStar from MicroPro Inc. The line editor EDLIN, which comes with PC-DOS, can be used to enter the source examples given in the text; however, a full screen editor will make source entry and editing easier and faster.

All the examples in this book were developed under MS-DOS 2.0, a system with 256K of RAM, two disk drives, and the Macro Assembler from IBM. The examples themselves are simple enough for first time ALP programmers. Should you want to experiment by using different or more sophisticated programming examples, by my guest. It is the only way you'll become proficient in writing your own programs.

ALP

A language is nothing more than a form of communication; a vehicle through which ideas, concepts, and actions are expressed. Suppose for a moment that you speak the language known as English, and you are trying to communicate in Spanish. Without knowing the language you can make grandiose gestures, draw pictures, or try some other form of audio/visual maneuvering to convey your meaning. The easiest solution is to find someone who speaks and understands both languages, English and Spanish. If the translation process goes smoothly and correctly, your meaning will be conveyed to the other party.

Similarly, when you program in AL, you program in the language of the assembler. What you type or enter from the keyboard is a language unique to the assembler for a particular microprocessor. The assembler statements are referred to as source code. The hitch is that the computer understands what is known as machine code, which is in the form of binary ones and zeros (more on binary in Chapter 1). You are faced with a similar problem; you need a translator.

The assembler translates your source program into a machine readable format that the computer can understand and execute. Should you mistype or misspell an entry, the assembler usually generates an error message. If, however, you enter a valid statement, but the statement is not what you intended, the assembler will not generate an error messsage, and the error may not be found until you try to run the program. The intent is not translated properly; only the process described in your Assembly Language source statement is translated.

This translation process is what must occur whenever you program in Assembly Language. The assembler translates your source programs into what is known as executable or object code; the binary ones and zeros the microprocessor understands.

The Definition of NEW

Learning AL does not happen overnight, nor does it occur through osmosis. Whenever you write a NEW AL program, remember that NEW is an abbreviation for Nothing Ever Works the first time. You will most likely find that a few bugs have crept into your program. These errors are common to all programmers and not restricted to just the beginner's code. More often than not your logic will be correct, but you'll have made a syntactical error that the interpreter translated literally when translating your source code.

There is one bright note in making mistakes; you learn from them. After spending considerable time poring over a section of code to find an error, you'll find yourself quickly learning and understanding the code in its entirety. While the program examples in this book have been debugged for you (by yours truly), I'm sure that a few typos will creep into your program as you enter the programming examples.

Hardware

While it is important to understand the hardware in your PC, I will limit my discussion to the microprocessor (8088) and a few of the support chips found in the PC. I will not go into a great amount of detail, as it is not the purpose of this book to delve into hardware specifics about the machine. The section will be more to introduce you to the world where Assembly Language programs must directly deal with such devices and program their operation. The main thrust of this text is to demonstrate how programs are written in the PC-DOS environment.

This implies that the operating system is responsible for initializing the support chips and in carrying out Input and Output (I/O) through these chips. The advantage inherent in presenting an introduction to ALP in this manner is that the programs you write will be transportable to another MS-DOS or PC-DOS IBM compatible computer. Should you attempt to directly program these chips, you may find that the physical memory or port address where they reside is not the same in all computers, even though the computers are said to be compatible.

Once you feel comfortable programming the PC, you'll be ready to learn more about programming hardware devices such as a communications interface chip (Intel's 8251A), a programmable timer such as the Intel 8253, or a peripheral interface chip such as the Intel 8255.

Chapter Review

Chapter 1 is an introductory chapter describing the different numbering systems used in assembler programming. Chapter 2 is an introduction to concepts that apply to all microprocessors and assemblers. Towards the end of the chapter I'll begin to focus directly on the 8088 microprocessor with a programming example that represents your first AL program (if you're new to ALP, of course).

Chapter 3 looks at the 8088 architecture in detail. Registers and their usage are defined and programming examples provided. Chapter 4 discusses the instruction set of the 8088 and the manner in which data are accessed (addressing modes).

Chapter 5 discusses features of the IBM Macro Assembler, MASM. The main emphasis of this chapter is using the many advanced features of the IBM Macro Assembler to control the assembly process. Chapter 6 describes MSDOS and macro instructions in detail: What they are and how to use them. A complete macro library is included on the diskette that accompanies this book. The macro library contains macro definitions for all of the functional calls supported by MS-DOS 2.0. Many BIOS macros have also been included.

Chapter 7 demonstrates how a program is written from start to finish. Chapter 8 discusses disk programming. The chapter contains four programming examples, including a program that reads, sorts and displays all filenames found in a diskette's directory.

Chapter 9 focuses on BIOS, the Basic Input/Output System of the PC. Programming examples include graphics and sound routines. Chapter 10 discusses telecommunications and presents an interrupt driven telecommunications program. The program supports disk and printer spooling while on-line and communcation baud rates up to 9600 baud.

For those of you who are experienced at writing AL programs for the 8088, you may want to skip directly to Chapters 6 through 10, which provide the necessary information required to write AL programs on the PC. All the information necessary to write programs that accept data from the keyboard, disk, and other I/O devices is presented here. There is also information on the operating system, MS-DOS, in these chapters.

I hope you find this book instructional and refer to it from time to time as one of your reference books on the IBM PC. Should you wish to communicate with me, you can write me:

c/o Argonaut Systems
POB 2492
Northbrook, IL. 60062

Please include a self-addressed stamped envelope if you expect a reply. You can also leave me electronic mail on CompuServe. My user I.D. number is 71625,121. I hope to hear from you.

1

Numbering Systems

It would be nice to have a computer that understood every word (spoken or typed), interpreted its meaning, and understood our numbering system. If that were the case, programming would be a snap. Nearly anyone could walk up to the machine, put it into a learning mode, and program it in plain English. If this were the case, there would be no need for this book or others like it. However, these are the machines of the future, ones that today's science fiction is made from.

Current technology is available that allows machines to talk and to listen to the spoken word. However, these voice input and output devices, like the printer or disk drive attached to your computer, receive and transmit information to and from the computer system via two discrete voltage levels. The voltage is either ON or OFF.

For the time being, computers understand only electrical signals that are either on or off. The language the computer understands is in binary form, the base two numbering system. Therefore, it is imperative that you understand binary and other numbering systems that are commonly used in programming, before you begin to look at Assembly Language programming on the PC. There is not one programming application that I can think of where a basic understanding of the different numbering systems is not important.

I'll examine numbering systems in this chapter and how data are represented using binary and hexadecimal notation. A later chapter provides you with a program that can be run on the PC to convert values among the decimal, binary, and hexadecimal numbering systems.

Numbering Systems

Binary (Base 2)

Humans chose a numbering system for their everyday use that was convenient to them—decimal, or base 10. As humans have ten digits (10 fingers), we chose a numbering system which also has ten digits—0 through 9. This is convenient for humans, but a microprocessor understands only the presence or absence of an electrical signal.

Since the microprocessor is concerned with only two possible states (ON or OFF), the binary numbering system could be used to represent the two possible states as a binary 1 for the ON condition and binary 0 for the OFF state.

This is the smallest amount of binary information in a digital system, and it is referred to as a BIT (BInary digiT). Several bits are usually grouped together to convey more information than can be found in a single bit. When 8 bits are grouped together, a *byte* is formed. A *word* is a grouping of the maximum number of bits the microprocessor can store internally. For the 8088, a word consists of 16 bits, the maximum width of any 8088 register (an internal storage location).

Counting in Binary

Examine for a moment how a decimal number is constructed. The number three-hundred-and-thirty-five is represented in decimal as 345. There is a one's position, a ten's position, and a hundred's position. Each digit's position is a multiple of the numbering system's base. The first position is 5 × 10 raised to the 0 power. The next digit has the significance of 4×10^1. The hundred's digit is expressed as 3×10^2. Each successive digit's significance is increased by a factor of ten.

Similarly, the binary number 1 0 0 0 0 1 1 1 represents the decimal number 135. The bit positions take on the following significance:

```
MSB (Most Signifint Bit)             LSB (Least Significant Bit)

b7   b6   b5   b4   b3   b2   b1   b0

1    0    0    0    0    1    1    1    Binary

128  64   32   16   8    4    2    1    Decimal

 7    6    5    4    3    2    1    0
2    2    2    2    2    2    2    2   Powers of two
```

The decimal values for all bit positions containing a 1 are added together to arrive at the decimal number being represented. A byte of all ones would equal the decimal value of 255. Since a byte of all zeros is possible, there are 256 total bit combinations possible within a byte (8 bits). What would the decimal value of the binary number 1 0 1 0 1 0 1 0 be equal to? Figure 1-1 shows the bit significance in a byte.

Bit Significance in a Byte

		BYTE							
		High Order Nibble				Low Order Nibble			
Hexadecimal Weight	→	8	4	2	1	8	4	2	1
Decimal Weight	→	128	64	32	16	8	4	2	1
Binary Weight	→	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Bit Position	→	B7	B6	B5	B4	B3	B2	B1	B0
Byte #1	→	0	1	0	1	0	0	1	0
Byte #2	→	1	1	1	1	0	0	0	0
Byte #1 (decimal)	=	0 + 64 + 0 + 16 + 0 + 0 + 2 + 0 = 82 which equals 52 HEX							
Byte #2 (decimal)	=	128 + 64 + 32 + 16 + 0 + 0 + 0 + 0 = 240 which equals F0 HEX							

Figure 1-1 A total of 256 possible states may be represented in a byte of binary. Shown are two bytes of binary. Notice that to represent the byte in hexadecimal, the byte is divided into two 4 bit quantities known as "nibbles".

Table 1-1 shows the binary numbers for 0 through 255 decimal. Notice that each time a bit position that is already set to a binary 1 has one added to it, the digit changes to zero. A carry is generated into the next bit position. This occurs in decimal also. It happens whenever 1 is added to 9:

```
9 + 1 = 10, 39 + 1 = 40, 99 + 1 = 100 etc.
```

Table 1-1
8 Bit Binary Examples

Decimal	Binary	Hexadecimal
0	00000000	00H
1	00000001	01H
2	00000010	02H
3	00000011	03H
4	00000100	04H
5	00000101	05H
6	00000110	06H
7	00000111	07H
8	00001000	08H
9	00001001	09H
10	00001010	0AH
11	00001011	0BH
12	00001100	0CH
13	00001101	0DH
14	00001110	0EH
15	00001111	0FH
16	00010000	10H
..	....	..
..	....	..
..	....	..
250	11111010	FAH
251	11111011	FBH
252	11111100	FCH
253	11111101	FDH
254	11111110	FEH
255	11111111	FFH

Adding numbers in binary is just as straightforward.

```
Binary                                    Decimal

    b7   b6   b5   b4   b3   b2   b1   b0

    -    1    1    -    1    1    1    1            Carries (- = no carry)

    0    0    0    1    1    0    1    1       27

+   1    0    1    1    1    0    1    1     +187

--------------------------------    -----

    1    1    0    1    0    1    1    0      214
```

Bit 1 (b1) in each of the two values to be added is set to binary 1. When they are added together, they produce binary 0 and generate a carry out of bit 1 into bit 2. But the carry into bit 1 from the previous addition (b0) must also be added. After adding the carry bit to the binary ones occupying the b1 position, the result is binary 1, with a carry into the next bit position, b2.

Binary Subtraction

To subtract two binary numbers, the concept of two's complement notation must be introduced. In the previous discussion, I spoke only of 8-bit unsigned numbers, with a range of 0 to 255. I'll now introduce the manner in which signed numbers, positive and negative, are represented. If you're concerned only with unsigned numbers, all eight bits of a byte or all sixteen bits of a word are available to represent a given value. If you want to add +3 to −1, that is to say subtract 1 from +3, the value of −1 must be represented within the 8-bit data field. This is accomplished by using the MSB of the data field to represent the sign bit. For byte values, a 1 in the MSB position indicates that the value of b0 through b6 is negative, while a zero in the MSB indicates that the value is positive.

Since the sign bit occupies one bit position, there are only seven bits (fifteen bits for a word) left to hold a binary value. Therefore, the range of signed values that can be represented in a byte is:

```
Positive Values                  Negative Values

0 0 0 0 0 0 0 0 (decimal 0)      1 1 1 1 1 1 1 1   (decimal -1)

      to

0 1 1 1 1 1 1 1 (decimal +127)   1 0 0 0 0 0 0 0 (decimal -128)
```

It is often convenient to depict positive and negative ranges by using a number line as follows:

```
                  8-Bit Byte Values

0..............128...............256     Unsigned Binary

-128............0...............+127     Signed Binary

                 16-Bit Word Values

0..............32,768...........65,536   Unsigned Binary

-32,768.........0.............+32,767    Signed Binary
```

I can hear you saying, "How the heck does a byte of binary one's equal minus one?" Negative numbers are represented in what is known as two's complement notation. To perform a two's complement on a binary number, flip all the bit positions and add a binary 1. If the bit position was a 1, change it to zero. If the bit position contained a zero, change it to 1. This forms the one's complement of the binary value. Do this for all the bit positions and add 1 to the result to form the two's complement. Actually, the value to be converted is subtracted from zero to form the two's complement, but complementing the number and adding 1 also works. The following example illustrates how this is done.

```
Number to convert:     0 0 1 0 0 0 1 1   (35 decimal)

One's Complement:      1 1 0 1 1 1 0 0

Add a binary 1:        1 1 0 1 1 1 0 1   (-35 decimal)

(Form 2's complement)
```

To discover what negative number a signed binary field contains, perform a two's complement on the number as follows:

```
Unknown Value:         1 0 1 1 0 1 0 1   (-? decimal)

Complement:            0 1 0 0 1 0 1 0

Add a binary 1:        0 1 0 0 1 0 1 1   75 decimal (Negative 75)

Convert binary to decimal --- >>          -(64 + 8 + 2 + 1)
```

By representing negative numbers in two's complement, a subtraction is carried out by adding the two values: (+A) + (−B). Let's say I want to subtract 2 from 1. I would first perform a two's complement on the subtrahend, then add that value to 1.

```
1) Convert +2 to -2.

Subtrahend:            0 0 0 0 0 0 1 0 (+2)

Complement:            1 1 1 1 1 1 0 1

Add  1 :               1 1 1 1 1 1 1 0 (-2)

2) Add -2 to 1.

                       0 0 0 0 0 0 0 1 (+1)

Subtrahend:          + 1 1 1 1 1 1 1 0 (-2)

                     ---------------------

                       1 1 1 1 1 1 1 1 (-1)
```

Check the answer by performing a two's complement on the result, keeping in mind that when the number is read, it has a negative magnitude.

```
Result:                1 1 1 1 1 1 1 1  (-1)

Complement:            0 0 0 0 0 0 0 0

Add  1 :               0 0 0 0 0 0 0 1  The value is 1 (negative)
```

A byte of all 1's is equal to a −1, and all subtractions can be carried out through addition.

Converting Decimal to Binary

To convert a decimal value to binary, divide the decimal value by 2. Write down the remainder. If the quotient is zero, you are finished with the conversion. If the quotient is not zero, repeat the process using the quotient as the dividend.

For example, convert 137 to binary:

```
Step         Quotient          Remainder
1) 137/2 =    68      1        Least Significant Bit.
                                         ^
2) 68/2  =    34      0                  |
                                         |
3) 34/2  =    17      0                  |
                                         |
4) 17/2  =     8      1                  |
                                         |
5) 8/2   =     4      0                  |
                                         |
6) 4/2   =     2      0                  |
                                         |
7) 2/2   =     1      0                  |
                                         |
8) 1/2   =     0      1        Most Significant Bit
```

The binary representation of 137 is:

```
MSB <------------------- LSB

B7  B6  B5  B4  B3  B2  B1  B0

1   0   0   1   0   0   0   1     BINARY
```

Hexadecimal Numbering (Base 16)

I have purposely limited the discussion on binary values to byte-wide data fields to keep the values used in the examples small. Binary notation is very inconvenient when larger values must be represented. A 16-bit value of 2498 decimal would be represented as 0000100111000010. It's harder to read and harder to work with large numbers in binary form. The number can better be represented using a numbering system that retains some similarity to the bit positions of the data field. Decimal does not meet our requirements, as there is no direct correlation between the decimal and binary digits.

Hexadecimal is most often used to represent binary data. Hexadecimal digits are comprised of the digits 0–9 and A–F. By dividing the data field into 4-bit slices called NIBBLES, you can represent a 16-bit binary number with four hexadecimal digits, and an 8-bit number with two hexadecimal digits. The decimal weights 8, 4, 2, 1 are given to each bit position of the nibble. If a nibble's value exceeds 9, the letters A–F are used to represent the decimal values of 10–15.

The number 2498 decimal is represented in binary and hexadecimal as follows:

```
0 0 0 0    1 0 0 1    1 1 0 0    0 0 1 0    Binary

8 4 2 1    8 4 2 1    8 4 2 1    8 4 2 1    NIBBLE bit values
| | | |    | | | |    | | | |    | | | |
|_|_|_|    |_|_|_|    |_|_|_|    |_|_|_|
 \|/        \|/        \|/        \|/
  0          9          C          2        Hexadecimal

    3          2          1          0
 16         16         16         16        Powers of 16

 0 x 4096 + 9 x 256 + 12 x 16 + 2 x 1 = Decimal 2498
```

The hexadecimal representation is therefore 09C2H. The H suffix denotes that the number is expressed in hexadecimal notation. In Chapter 7, I'll show you how to write a program which accepts decimal input from the keyboard and converts the number to hexadecimal and binary and displays the result on the video screen.

When hexidecimal notation is used in your source program, you'll need to remember this next rule. When the hexadecimal digit begins with a letter (A–F), you must precede the number with a zero. For example, if you wanted to use the hexadecimal number FADEH, the assembler would not be able to determine whether the characters represented a constant or a name. The leading zero informs the assembler to treat 0FADEH as a number and the lack of a leading zero (as in FADEH) informs the assembler that the characters represent a symbolic name.

Addition of Hexadecimal Numbers

As in the binary and decimal numbering systems, hexadecimal digits can be added. When the addition exceeds a digit's capacity, the digit rolls over to zero and a carry is generated into the next digit position. For example:

```
  1---          <<---- {Carries}

  9FC0 H        <<----|
                      |
+ 1938 H        <<----|
---------
  B8F8 H        <<---- {Result}
```

Binary Coded Decimal

Often you need to store and work with numbers in a computing system that are encoded in Binary Coded Decimal, BCD. BCD uses 4 bits to represent the decimal digits 0–9. A byte can therefore contain two packed BCD digits. If a byte contains only one BCD digit (unpacked), the other nibble must be zeroed. For example:

```
Unpacked BCD                 Packed BCD

0000 0010 = 2 (BCD)          1001 0001 = 91 (BCD)

0000 1001 = 9 (BCD)          0110 0111 = 67 (BCD)
```

The following are illegal representations:

```
0000 0010 = 0A H - Is not a digit 0-9.

1110 1100 = EC H - Neither nibble is a digit between 0-9.
```

Since BCD is used to represent the decimal digits 0–9, any 4-bit combination in excess of the decimal value 9 cannot be considered a valid BCD digit.

BCD Addition

When the microprocessor adds numbers, it treats all values as binary. It does not know or care if the binary values being added are BCD encoded or in some other encoded format. The computer adds the BCD digits just as if they were binary numbers, as follows:

```
  0101 0011 = 53 BCD   {83 decimal (Binary weight)}

+ 0010 0110 = 26 BCD   {38 decimal (Binary weight)}
--------------------
  0111 1001 = 79 BCD   { 53 + 26 = 79 BCD }

                       {121 decimal (Binary weight)}
```

The answer is correct:

```
  0011 0111 = 37 BCD    {55 decimal (Binary weight)}
+ 0011 0010 = 32 BCD    {50 decimal (Binary weight)}
-----------------
  0110 1001 = 69 BCD    { 37 + 32 = 69 BCD }
                        {105 decimal (Binary weight)}
```

The answer is not correct:

```
  0011 1001 = 39 BCD    {57 decimal (Binary weight)}
+ 0011 0001 = 31 BCD    {49 decimal (Binary weight)}
------------------
  0110 1010 = 6AH {NOT A VALID BCD DIGIT!}
                        {6AH= 106 decimal (Binary Weight)}
```

The reason this last example is not a valid BCD digit is that the low order nibble produced a binary combination in excess of 9. What it all means is that there has to be a method to correct for such a condition. What would happen if you added 6 to the low order nibble that was in error? Let's see what result this would produce.

The original addition was to produce a result of 70 in BCD format (39 + 31). The actual result was 6AH.

```
                0110 1010 = 6AH
Now add 6 -->)  0000 0110 = 06 {BCD}
               ------------------
                0111 0000 = 70 {BCD}
```

The answer is now correct, and it is represented in a BCD format. The rules are simple. If the result of a BCD operation produces a value in excess of 9, add 6 to the result to correct the result to proper BCD format. There is still one more case to be considered. What would happen if the result of the addition produced a nibble whose value has passed right through the range of 10 to 15, which is not allowed?

Consider the following:

```
  0000 1001 = 09 {BCD}

+ 0000 1001 = 09 {BCD}

-----------------------

  0001 0010 = 12 {?}
```

The result looks like BCD, but it isn't correct. We passed through the range necessary to effect an adjustment of the nibble, when the nibble's value exceeds 10. The former rule cannot apply since 2 is less than 10. However, there has been a carry generated out of the low order nibble (bit 3) into the first bit position of the high order nibble.

Let's qualify the former rule: Whenever the result of a BCD addition causes a digit's value to exceed 10, OR when a carry is generated out of a nibble due to the addition, add the value of 6 to the nibble to correct the result. When this is done, you obtain 18 BCD, the desired result of 9 plus 9. The important point to remember is that the CPU does not know or care if the number to be added is in BCD format. To the CPU, all forms of character encoding appear in binary form. Therefore, it is the programmer who makes the distinction of how the arithmetic should take place and what graphic characters and numbers the binary groupings are to represent.

Character Encoding

ASCII

In an attempt to standardize and formalize the manner in which computers exchange information, a method to encode alphabetic, numeric, and special characters was devised. ASCII (American Standard Code for Information Interchange) is a 7-bit code that defines the individual bit groupings for each letter of the alphabet and the ten numeric digits defining the decimal numbering system.

The eighth bit is used to transmit parity, a form of error checking that is dependent on the total number of binary 1's contained in the character transmitted. If the total number is even, and you are using even parity, the eighth bit is zero. If the total number is odd, the eighth bit is set to 1, to make the total number of 1 bits an even number.

In total, there are 128 different bit groupings ($2^7 = 128$) assigned to represent character, number, and special character graphics as well as special control codes. The control codes are used to control how the communication between two computing devices takes place. For example, the binary bit combinations of 001 0001 (DC1 = 11H) and 001 0011 (DC3 = 13H) are commonly used to resume or suspend communications between two devices. These controls are commonly referred to as XON and XOFF, for transmit on, or transmit off.

Although the programmer determines how the bit combinations the computer is acting upon at any given moment are to be interpreted, ASCII was developed to insure a standard means of transferring information between computing devices. In other words, if I know before computer-to-computer communication begins, that your terminal will be transmitting ASCII text to mine, and I receive the bit combination 100 0001 (41H), I know that the bit combination will always be used to represent the letter A. Similarly, if I transmit the bit combination of 011 0011 (33H) to your terminal, you will always interpret this bit combination as the character 3.

Other Binary Codes

This would be fine if ASCII were the only coding used to represent letters, numbers, and special characters, but it is not. There are other binary codings which are used in international and mainframe computer communications. EBCDIC (Extended Binary Coded Decimal Interchange Code) is widely used in the IBM mainframe world. EBCDIC (pronounced EB-SID-IC) is an 8-bit code used to represent control, alphabetic, numeric, and special characters. The EBCDIC code can be found in Appendix A.

Baudot Code

Baudot is another binary code used to communicate information between machines. Baudot is commonly found in international and domestic communications involving telegraph circuits. It is a 5-bit code that is usually associated with teletypewriters and teleprinters. Typically, these devices have limited communications speed (usually less than 110 bits per second). By using only 5 bits to encode a character rather than the 7 bits required in ASCII or the 8 bits required in EBCDIC, characters are transferred between communicating equipment faster. The Baudot character set can be found in Appendix A.

I'll discuss ASCII and other binary codes further when I explain the 8088's instruction set. The 8088 microprocessor has some powerful instructions that benefit the

Assembly Language programmer who must work with ASCII and BCD characters or strings. But for now, let's turn our attention to the language of assembler.

Chapter 1 Review

1. The role of an assembler is to ___________ a ___________ file into an object file.
2. In binary there are ___________ possible states:

   ___________ and ___________.
3. The computer subtracts two binary numbers using ___________ complement notation.
4. Give the one's complement form of the following binary numbers:

   A. 01010110 C. 10111110
   B. 11110000 D. 01010101
5. What is the two's complement of the binary numbers in problem 4?
6. Convert the following binary values to hexadecimal notation:

   A. 10011110 C. 11111111
   B. 11100111 D. 1000110000001111
7. Convert the following hexadecimal numbers to binary:

   A. 0FFA1H B. 2078H
   C. 7FFCH D. 0003H
8. Why does the hexadecimal number in 7A above contain a zero prefix?
9. Two popular types of character encoding are ___________ and ___________.
10. The base of each numbering system is:

    A. Binary is base ___________.

    B. Hexidecimal is base ___________.

    C. Decimal is base ___________.
11. What is wrong with the following hexadecimal number?

    0FG77H

2

Assembly Language Programming

This chapter introduces novice programmers to the discipline required to be an Assembly Language (AL) programmer. Many new programmers feel that there is something mystical about learning to program in Assembly Language. If there were something mystical about the language, I'm sure more programmers would be named Merlin. Since I know only one programmer who calls himself Merlin, and his programs are far from being mystical, it is safe to assume that most of us mere mortals who really have the desire to program in Assembly Language will be able to do just that by the end of this book.

Like all good things worthy of personal possession, the skills required to become an AL programmer are not acquired without a certain degree of frustration. However, the rewards can be considered substantial, in personal enjoyment and financially, if you are so inclined.

The Programmer

The fallacies surrounding ALP are as numerous as those surrounding the programmer. A typical characterization is as follows: ALP programmers must possess a high degree of mathematical skills, and they must also hold higher level degrees. They are hermits who let nothing stand in the way of finishing the project at hand, knowing full well that software projects are rarely finished, they are merely released.

If married, you have to pity the poor spouse of the AL programmer. Dinner conversations revolve around bits and bytes, algorithms and code. Shopping with such an individual can be almost as exciting, as every book store is an invitation to "keep current" with the technology. Computer stores are sought out with a most unhealthy, if not obnoxious zeal.

Contrary to the stereotype presented, a higher level degree and advanced mathematical skills are not required for an individual to become proficient at AL programming. Although the skills certainly wouldn't hurt, they are not required. When an application requires a tremendous amount of computational power, the program is usually written in a high level language such as FORTRAN (FORmula TRANslation language). It's rare when such an application would be coded totally in Assembly Language.

As for the AL programmer being a hermit, nothing could be further from the truth. Most commercial software projects are developed by a team of software engineers in a team environment. An individual programmer has responsibility for a specific portion of code which is later incorporated into a main program consisting of the modules and subprograms created by the other individuals. The team is usually composed of specialists. There may be one individual who is proficient at data communications while another is concerned with internal data structures or the user interface.

Obviously, if you are programming your own personal computer, you will most likely be doing so without the aid of other programmers; hence, the purpose of this book. Books are often your only resources when working on a section of code. The modular programming concepts of a program written in a team environment still apply. However, you are the one who is responsible for writing all the modules which ultimately become the finished program.

Those who do become proficient at AL programming tend to view the world and problems in a somewhat different perspective than those engaged in other professions. The reason may be traced to the extremely logical thought process forced upon the AL programmer by the very nature of his/her chosen career. There is a tendency to apply that logic to the world and those around us. The unobvious becomes the obvious, and problems become puzzles which are challenges to solve and master. It is all a matter of attitude and a learned set of skills.

One thing to keep in mind is that you cannot program the computer in any language, if you yourself do not understand the application. If you cannot explain the problem and define a method to arrive at the solution, the computer will be of little help. You can be the most creative and productive programmer on your block and not be able to write the program for a problem beyond your own expertise.

For example, if you have never flown an airplane, it would be very difficult if not impossible to write a believable flight simulation program. For someone else who is involved with aviation and who can program in Assembly Language, writing such a program will be easier than for most of us. This is what makes programmers so valuable; it is not that they know how to program in Assembly Language, but rather that they understand an application and can define its solution through Assembly Language.

Language Differences

Once you master the assembler's language for one microprocessor (also referred to as a CPU, or Central Processing Unit), you'll be able to transport your skills to a different microprocessor. The process does require learning a new architecture (registers, status flags, etc.) and a new instruction set; however, the basic and fundamental concepts of programming in assembler apply to any microprocessor. For example, to load the accumulator of the 8-bit 6800 (Motorola) with a decimal value of 16, you would enter the following statement into your source program:

LDAA #16. The Zilog Z-80 CPU requires LD A,16. To accomplish a similar function on the 8088/86 CPU, you would type: MOV AL,16. Notice that one microprocessor's assembler requires the instruction LD (LoaD), and the other requires a MOVe instruction. The difference between these particular instructions is trivial and easily learned. Other instructions may not be so obvious and may require closer inspection.

Computer Languages

A computer language can be classified as either being a high level language, such as BASIC, FORTRAN, COBOL, or Pascal, or a low level language, such as Assembly Language. The classifications can be further refined as to whether a high level language is interpretive or compiled. Programming languages exist only as an aid in describing the steps necessary to solve a problem or a set of problems. Ultimately, all but the interpreters produce the machine code the computer understands, and, as such, they require a compromise as to the execution speed and the size of the machine code each will produce.

Higher Level Languages

The assembler represents a quantum leap over programming a machine directly in machine language, yet the human interface is still lacking. It's more desirable to increase a programmer's efficiency by providing a programming language that uses English-like statements to define the problem you want to solve. This is beneficial for several reasons.

First and foremost, the programmer could be isolated from the hardware specifics of the machine and not be required to learn a thing about what's "under the hood" of the computer. It's a little like the old adage of not having to be a mechanic to know how to drive a car.

"Who cares about the registers anyway? Why should I care about an AX or BX or XYZ register when all I want to do is print 'Hi Sue, HAPPY BIRTHDAY! I Love You' on the screen of my computer?" You might ask. Actually, there is no reason to learn Assembly Language to have the computer print a simple message on the screen. The simple BASIC statement: PRINT "Hi Sue, HAPPY BIRTHDAY! I Love You." accomplishes what will take perhaps hundreds of Assembly Language statements to do—display the message on the screen of the computer. By using a high level language, you are isolated from the hardware specifics of the system. This allows you to concentrate on solving applications problems.

Interpretive Languages

An interpretive language, such as the BASIC interpreter supplied with many home computers, must interpret each program statement prior to executing it. Is it a valid statement? Is its syntax correct? After properly identifying the statement as valid, the actual code which performs the statement's function is executed. However, it must be noted that this interpretation occurs each and every time the statement is executed. Due to the overhead involved, programs execute slower than compiled programs or Assembly Language programs.

The interpreter occupies memory along with the application program being run. Because both the interpreter and the application program must be in memory at the same time, the physical size of the program is also greater than one written in AL.

Compiled Languages

Compiled languages fall somewhere between interpreted languages and Assembly Language, in that compilers take on certain characteristics of both. Source programs are written in a high level language and translated (much like Assembly Language) into machine code by the compiler. The object code generated by the compiler tends to execute faster than an interpreted program, but it may not be as fast as a program written in Assembly Language. The program's size is, in most cases, smaller than that of an interpreted program, yet is larger than a similar program written entirely in Assembly Language. Compilers exist for a variety of languages, such as BASIC, FORTH, Pascal, COBOL, and the C language.

With the many language options available, you may still find it difficult to find a language to efficiently solve all problems in a given application. The reason is that no one language suits all applications. FORTRAN is useful in solving scientific oriented problems. COBOL is widely used in business applications. Pascal's claim to fame is that it forces a programmer to use structured programming techniques and is of great use in teaching these concepts. BASIC is of interest because it is widely used and is usually the first computer language an individual learns to program in. For real-time applications where execution speed and memory conservation are of prime consideration, Assembly Language is the choice.

Machine Language

When computational machines were first conceived, there were no languages that the machine's programmer could use to instruct the machine what it should do. Therefore, each command or machine instruction had to be manually entered. The process was slow, tedious and extremely error-prone.

Machine language is composed of definite digital patterns of binary one's and zero's:

```
Machine Language                 Assembly Language

10110100  00001001               MOV AH,9
```

In the above example, machine language, the language a microprocessor understands, is composed of finite combinations of binary ones and zeros. The microprocessor decodes these patterns, and performs the specified function. In this example, the internal register AH (a storage location inside of the 8088) is loaded with a value of 9 decimal.

A grouping of 8 Binary digITS (BITS) is referred to as a byte of information. A grouping of 2 bytes, or 16 bits, is referred to as a word. A typical microprocessor sequentially fetches these binary values from memory and decodes them into meaningful instructions. As each pattern represents a particular instruction to the microprocessor or data which is required by the microprocessor to properly execute the instruction, it is extremely important that the binary values be placed in memory and remain in memory without alteration.

For example, the MOV instruction used in the previous example might appear in memory as a binary sequence of bits:

```
         1  0  1  1  0  1  0  0  0  0  0  0  1  0  0  1

Bit #   b7 b6 b5 b4 b3 b2 b1 b0 b7 b6 b5 b4 b3 b2 b1 b0

              1st BYTE                2nd BYTE
```

This sequence of bits tells the microprocessor to MOV the value of 9 into the AH register of the microprocessor. The machine language program sample shown below depicts a short program as it would appear in memory and in binary form. Should a bit be lost from the machine instruction, either the value moved to AH will change, or the instruction itself will change (it will be decoded as a different instruction) and render your program useless. It is very important that each instruction be entered into the machine accurately.

```
101110010000000000011000            MOV CX,24

111010000000000100100010      CLS:  CALL CRLF

1110001011111011                    LOOP CLS
```

Only the binary one's and zeros are stored in memory, not the English-like abbreviations to the right. The routine above is part of a larger program used to clear the video display of the IBM. Imagine if you had to enter each one and each zero of the larger program (see Listing 2-1) into memory manually!

As each machine instruction directly causes an action to be taken by the microprocessor, machine language programs execute the fastest of all language types. The speed at which an instruction executes is of great importance in many time-critical applications and is dependent on the system clock that drives the microprocessor (Figure 2-1), and the manner in which the microprocessor's designers chose to decode a given instruction. Similar instructions executing on different microprocessors running at the same speed do not always execute in the same amount of time.

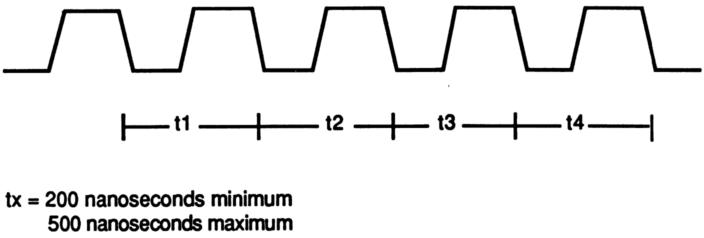


Figure 2-1 The system clock provides a regular "heartbeat" to the microprocessor. The clock is provided by another integrated circuit (IC), an Intel 8284 clock generator.

As a finished program can contain hundreds or thousands of machine instructions to solve a given problem, it would be a very tedious chore to code your program entirely in binary. Yet, without efficient programming tools, early computers were programmed in exactly this manner. Only a pure masochist liked programming in machine language, but someone had to do it in order to create the first assembler.

Assemblers

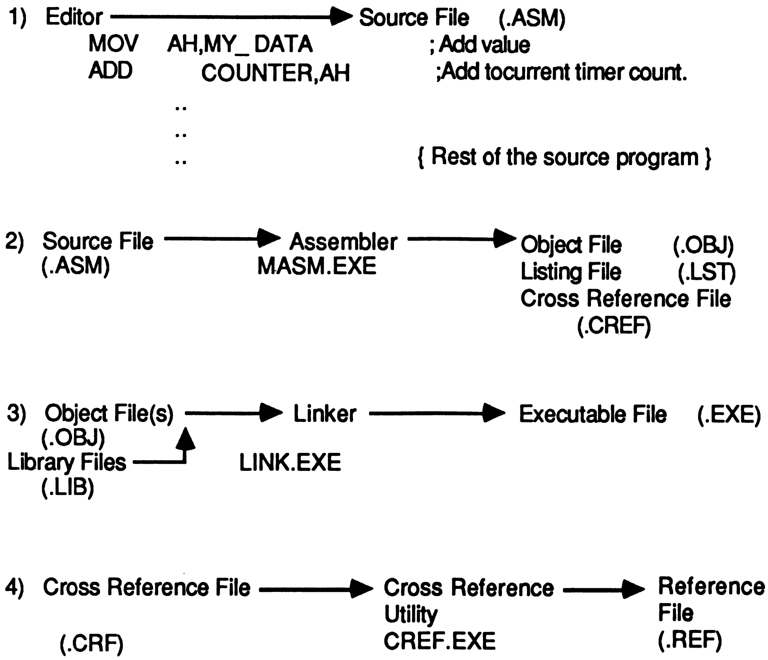
You would think that it would be much easier to code the same instruction in a language that was more suited to the human programmer, while maintaining the advantage of machine language execution speed. It would have to be a language that allowed the programmer to describe the desired action he/she wants the microprocessor to perform. The statements the programmer types at the keyboard create what is known as a source file. The statements are later translated to the machine code the microprocessor understands. Rather than setting sixteen or more switches on a computer's front panel and loading the values directly into memory, an assembler can translate the statements in the source file into the object (machine) code required.

An assembler is itself a program created to automate the many routine, repetitive, and error-prone tasks associated with programming a computer directly in machine language. The assembler program translates a source file of terse English-like abbreviations known as mnemonics into what is referred to as object code and/or machine code.

Figure 2-2 shows the four primary steps in creating and documenting your programs:

1. A program called an editor is used to create a source program.
2. The source program is translated to an object file by the assembler. The assembler optionally creates a listing and a cross-reference file.
3. The linker produces an executable file capable of being run on the IBM PC. Library files may also be used as input to the linker.
4. The cross-reference utility can then be used to create a cross-reference of all symbols appearing in the program and where in the program they appear.

Because there is a direct relationship between the AL source statement and the resultant object code, programs written in AL execute as fast as if the program had been entered into the machine in pure machine language. The assembler has removed the chore of coding the program in binary.



```
MOV   AH,MY_DATA          ; Add value
ADD      COUNTER,AH       ;Add tocurrent timer count.
        ..
        ..
        ..                { Rest of the source program }
```

1) Editor
Source File (.ASM)
2) Source File (.ASM)
Assembler MASM.EXE
Object File (.OBJ)
Listing File (.LST)
Cross Reference File (.CREF)
3) Object File(s) (.OBJ)
Library Files (.LIB)
Linker LINK.EXE
Executable File (.EXE)
4) Cross Reference File (.CRF)
Cross Reference Utility CREF.EXE
Reference File (.REF)

Figure 2-2 Assembly Language Source to Machine Language/Executable Code.

The Editor

Another program—called an *editor*—is used to create the source program or code. CP/M's™ program ED and MS-DOS's EDLIN are examples of line oriented editors. They allow you to create and edit one line of source code at a time. The line forms what is called a statement; MOV AX,16 moves the value of 16 into the accumulator of the 8088/86 microprocessor. When a particular line of code has been edited, you can begin editing another line.

A full screen editor is much like a word processing program, because a full screen of text (many source lines) is displayed. By using the cursor keys, you can move freely around within the source program. In fact, many programmers use word processing software to create their source code files. Combining a full screen editor with the advanced editing features of word processing software results in increased efficiency by decreasing the amount of time required to create and edit a source file. This is a vast improvement over simple-minded and primitive line oriented editing.

Symbolic Representation

One of the many complicated tasks the assembler program can do is to keep track of the different memory locations used for data and program storage. Figure 2-3 illustrates the use of symbols that represent data and memory locations, as well as constant numerical values required by the program. Without this ability, you would have to keep track of each memory location and data value referenced within the program.

For example, you may want to assign a symbol name to a numeric constant that will be used several times in your program. Rather than hard coding the value into the instruction, it is often more advantageous to assign the value a name at the beginning of your program, and then use the name rather than the value in the program. By assigning the label PTR_OUT to the value 03B8H, the program becomes easier to read and understand.

Another example of symbolic usage is in the assignment of a name to a port address. Port addresses are used to communicate with input and output (I/O) devices such as a printer, video display, or keyboard. Actually, the port addresses belong to the

```
            PAGE 55,132
        TITLE Example of Symbol Assignments
; ***************************************************************************
; Date: 06/05/84
; By: Gary A. Shade
; Last Revision: 06/05/84
;
; ©1984 by Gary A. Shade, All Rights Reserved.
;
; This program does a lot of nothing. That is, it is only to
; demonstrate symbolic usage in a source program.
; It also demonstrates where each assembler field originates, and the general
; form of an assembly language program.
;
; Entry Conditions:  Save DS then a word of 0000H on the stack.
                     Required by MSDOS.
; Exit Conditions:    Registers used within the program are altered.
; ***************************************************************************
;                               Definition of Assembler Fields
;  <__Label__>  <__Opcode__>  <__Operand__>  <__________Comments__________>
EX_A_STACK   SEGMENT      PARA STACK 'STACK'
                    DW   125 DUP(?)            ; Establish Stack Area
EX_A_STACK          ENDS
;
;
EX_DATA   SEGMENT   PARA 'DATA'
COUNTER1            DB          OFFH          ; 8 BIT COUNTER
COUNTER 2           DW          OFFFFH        ; 16 BIT COUNTER
COUNTER3            DQ          OFFFFFFFFH    ; 32 BIT COUNTER
EX_DATA   ENDS                                ;END OF SEGMENT
;
EX_CODE_SEG SEGMENT          PARA 'CODE'      ; Code Segment begins
ASSUME   CS:EX_CODE,SS:EX_A_STACK,DS:EX_CODE,ES:EX_CODE
START            PROC FAR                     ; Declare as 'FAR'
                    PUSH      DS              ; Save Code seg, passed
                                              ; by MSDOS
                    XOR       AX,AX           ; Clear AX
                    PUSH      AX              ; Save on stack.
                    MOV       AX,EX_DATA      ; Set up segment
                                              ; registers
                    MOV       DS,AX           ; Data Segment
                    MOV       ES,AX           ; Extra segment
                    MOV       CX,COUNTER2     ; Get loop delay factor
DELAY1:             LOOP      DELAY1          ; Loop until CX = 0.
                    MOV       CL,COUNTER1     ; Get byte delay count
                    XOR       CH,CH           ; Clear High Order byte
DELAY2:             LOOP      DELAY2          ; Execute LOOP until
                                              ; CX=0
                    MOV       CX,PTR WORD COUNTER3 ; Delay count
DELAY3:             NOP                       ; WASTE SOME TIME
                    NOP
                    LOOP      DELAY3          ; Execute last loop.
```

Figure 2-3 Example of Symbol Assignments

hardware, which interfaces the microprocessor to these devices. More than likely the port addresses given to these devices will vary from computer to computer. In order for the program to be run on another computer using the same microprocessor, these values and others will have to be redefined. By defining the port addresses symbolically, you need only change the line of code which assigns the label to a physical port number, rather than changing every occurrence of the port address within the program.

Assume the printer for a given computer is at port address 0E0H (the H means the number preceding it is given in hexadecimal), or 224 decimal. By writing a statement such as:

```
PRTOUT   EQU 0E0H
```

The printer output port (0E0H) is assigned via the EQU directive to the label (PRTOUT). There may be a hundred different places in your program where data is sent to the printer via port 0E0H. If a label were not assigned to the port, a hundred lines of code would have to be found, and the port's value changed if the system's hardware should change.

One last example of assigning labels within a program can be illustrated by using a label to specify a strategic entry point into a portion of the program. As an example, consider the following source code:

```
START:          PUSH DS          ;SAVE CODE SEG PASSED FROM MS DOS

                XOR AX,AX        ;MUST SAVE A ZERO OFFSET INTO

                                 ;CODE SEGMENT ON STACK.

                PUSH AX          ;SAVE WORD ON STACK

                CALL INIT        ;INITIALIZE THE SYSTEM

                JMP CLRSCN       ;CLEAR THE SCREEN
                  . .

                  . .    {MORE PROGRAM STATEMENTS}

                  . .

CLRSCN:         PUSH    DX       ;SAVE REGISTERS

                  . .

                  . .

                JMP     THERE    ;JUMP TO ANOTHER PLACE IN PROGRAM

INIT:           CALL    DISKINIT ;INITIALIZE DISK DRIVES

                CALL    WHOEVER  ;ETC.
                RET             ;RETURN TO CALLER

THERE:          MOV AL,PTROUT   ;GET PRINTER OUT PORT

                 . .     {The rest of the program}
```

Nowhere within the program were absolute addresses for the instructions labeled START, INIT, THERE, or CLRSCN defined. The assembler calculates the absolute memory addresses where each section of code resides when assembling your source program.

Labels are also more easily remembered than are absolute addresses and values, and they are more easily changed should the physical values change. The assembler remembers the values assigned to the labels used in the program and substitutes these values when the assembler translates the source file to object code.

Forward References

The IBM Macro Assembler makes two passes through the source file. On the first pass, the assembler builds a table of all the symbolic names defined in the source file. The assembler does not produce object code on this pass. The second pass produces the object code for the source statement based upon the information gathered during the initial pass.

During the first pass, the assembler may not know the exact location of a symbol it has encountered but not defined. This is known as a forward reference. It is a little like a writer who makes reference to an illustration while writing a book. Usually the reference reads something like "Reference Figure 1-XX". During a second pass through the manuscript, after the chapter has been written, the exact figure number is inserted. The process is similar to the first and second pass of an assembler through a source program. It is only during the second pass that the assembler knows the relative addresses and what values to use in place of the names and symbols encountered during the first pass.

Cross Assemblers

Cross assemblers are used to create object code for a microprocessor other than the one used in the computer the code is being developed on. There are several companies marketing cross assemblers that run on the IBM PC. By using a cross assembler, you can create programs for other microprocessors, such as the Motorola 68000 or 6800 series, the Zilog processors (Z-80, Z-8000), and other Intel processors, such as the older 8080 and 8085, on your PC. The programs created using a cross assembler cannot be executed (run) on the development system, since the machine language of each microprocessor is unique to itself. The object code must be downloaded, or transmitted, to the target system for execution.

Cross assemblers are of value to anyone who must learn or develop machine code for another processor and cannot justify the added expense of another computer system to do so. By using cross assemblers, one computer system can be used to create programs for many others.

Linkers

Previously I mentioned why it is a good idea to develop programs in a modular form; smaller chunks of code are easier to debug and understand. Their functionality is much more explicit than one large program. The purpose of the Linker is to join these smaller chunks of code together to produce a finished program.

You can think of a program as being analogous to a chain. A chain is not a chain until all the individual links are connected. In a similar manner, a program is not a program until all the individual modules have been linked together.

Program libraries containing dozens of small often-used routines or programs can be built and stored on disk. By using the linker and specifying which modules are to be linked together to form the completed program, you can create new programs without having to write the entire program from scratch. If, as an example, you have already written a routine that performs bounds checking on a set of numbers, and you need the same routine in a new program, simply include the bounds checking module in the linkage.

A relocatable object file as shown in Figure 2-4 is produced by the assembler. The relocatable expressions are those marked with an R next to the object code. The linker also resolves any relocatable expressions, including any values of data, constants, or memory addresses that are defined in other modules. When the program is loaded and executed, a loader program uses the information supplied by the linker, enabling the program to be loaded and executed anywhere in memory.

```
0039  C7  06 0000 R  0140  DRAW_IT2: MOV   STARTX,320
003F  C7  06 0002 R  00AA            MOV   STOPX,170
0045  C7  06 0004 R  0032            MOV   STARTY,50
004B  C7  06 0006 R  0096            MOV   STOPY,150
0051  C6  06 000B R  07              MOV   COLOR,W_B
0056  E8  01AC R                     CALL  DRAW_LINE
0059  E8  0242 R                     CALL  DELAY
005C  C7  06 0000 R  0113  DRAW_IT3: MOV   STARTX,275
0062  C7  06 0002 R  0177            MOV   STOPX,375
0068  C7  06 0004 R  0064            MOV   STARTY,100
006E  C7  06 0006 R  0064            MOV   STOPY,100
0074  C6  06 000B R  07              MOV   COLOR,W_B
0079  E8  01AC R                     CALL  DRAW_LINE
007C  E8  0242 R                     CALL  DELAY
007F  E8  0242 R                     CALL  DELAY
```

Figure 2-4 Example of Relocatable Code

Creating a Source Program

Syntax

You'll notice in Listing 2-1 (see Appendix D) that there are specific fields where you enter labels, instructions, operands, and comments to form a source statement. The obvious deduction is that a certain syntax must be followed in order for the assembly process to occur properly. The format in which the source code must be entered is:

```
Label          Operations-Code          Operand(s)          Comment.
```

Listing 2-1 should be referenced as I discuss how a source program is created.

Label Field

We have seen why labels or symbolic representation within the source program is important. The manner in which names are assigned is of equal importance. The IBM Macro Assembler allows you to create labels of up to 31 characters in length. This allows you to assign meaningful names to constants and program entry points. Not all assemblers allow you this flexibility. Labels can consist of any of the following characters:

1. Alphabetic characters: A through Z (lower- or uppercase).
2. Numeric digits: 0 through 9
3. Special Characters: $. _ ?

The IBM Macro Assembler manual states, "Labels can start with any character except a numeric digit. If a period is used in the label, it must be the first character of the label." Words and names reserved for the 8088 registers and instructions should also be avoided, as should words reserved by the assembler (referred to as directives) when assigning symbolic labels. Some examples of labels are:

Valid Label	Invalid. --->>	Comment.
MY_DATA	0MY_DATA	(Digit 1st character)
?ASCII_CONVERSION	?ASCII.CONVERSION	(Illegal use of .)
LOOP020	LOOP	(8088 instruction)

Additionally, a label may contain a colon suffix if the label is used to define an entry point into a section of code:

```
START:   PUSH DS     ;SAVE DS AND AN OFFSET OF ZERO
         XOR AX,AX
         PUSH AX
```

There is another use for the colon suffix, and that is to inform the assembler that the label is to be assigned a TYPE attribute of NEAR. NEAR attributes can be accessed only within a given segment. The absence of a colon informs the assembler that the label can be referenced from other segments, and a TYPE attribute of FAR should be assigned to the label.

Labels should not contain a colon suffix if the label is used with what is known as a pseudo-op. This is an assembler directive that instructs the assembler how to perform the assembly or what values to use during the assembly. For example:

```
PTR_OUT      EQU     0E0H          ;ASSIGNS 0E0H TO PRINTER OUT

PTR_BUFFER   DW      100 DUP (?)   ;DEFINES STORAGE

FLAG1        EQU     THIS BYTE     ;ASSIGNS FLAG1 TO TYPE BYTE
```

Furthermore, a label should not contain a colon suffix when it is used as an operand in an instruction:

```
MOV AL,FLAG1              ;MOVES FLAG1 INTO AL

MOV PTR WORD FLAG1,AX     ;STORE AX IN THE TWO BYTES

                          ;STARTING AT FLAG 1
```

A label is defined by entering its name in the first field of the source statement. Labels must originate in column one.

Operations Code (Op-Code) Field

Op-codes are the actual mnemonics that represent the instruction you want the microprocessor to execute. Op-codes must be entered in the second field of the source statement, separated from the label field by at least one space. To make the program more readable, programmers usually use tabs to align the fields of the source program.

An example of an op-code is JMP. JMP is a mnemonic which the assembler translates to a machine executable instruction.

When used, Pseudo-ops must also be entered in the op-code field. You'll notice in the previous examples, EQU, DW, and DB are all pseudo-ops that appear in the op-code field.

Operand Field

The operand field tells the assembler what it is we want to move (MOV) or jump (JMP) to and may require two operands, a source and a destination. An operand may be a constant (value), a register, or a label. Some instructions are register implicit and do not require any operands at all:

```
MOV AX,10H          ;TWO OPERANDS AX$azDESTINATION
                    ;USING A CONSTANT AS THE SOURCE
MOV AX,COUNT        ;MOVE THE CONTENTS OF MEMORY
                    ;LOCATION COUNT TO THE AX REGISTER
NEG AX              ;NEGATE (2'S COMPLEMENT) THE ACCUMULATOR
                    ;ONE OPERAND, AX = DESTINATION
JMP THAT_PART       ;ONE OPERAND, THAT_PART = DESTINATION
CLD                 ;CLEAR DIRECTION FLAG - IMPLIED OPERAND
                    ;NOT REQUIRED TO BE EXPLICIT.
```

The operand field must be separated from the op-code field by at least one space.

Comment Field

A comment must originate in the first column of the label field if the entire line is to be a comment, or it must be separated from the preceding field by at least one space. Comments must begin with a semicolon (;), as shown in the previous examples. All characters entered after the semicolon are ignored by the assembler and treated as a comment. Comments make your program more understandable to you and others (should they have the fortunate task of maintaining your code after you have moved on to greener projects).

Often, when you are writing programs, it is not practical to completely document your code as you write it. However, there should be enough comments to describe the code in a precise manner, so that it will be easy for you or someone else to understand the program today or a year from now. Once the code has been debugged and finalized, go back and more thoroughly document the program by adding more comments wherever possible.

I have a tendency to occasionally state the obvious in my comments such as:

```
JMP START    ;Jump to start.
```

Nothing infuriates me more than to read a source listing of a program which is documented like this. It infuriates me further when I sheepishly recognize the code as mine. Why did I jump to START? The reason is not always obvious unless you know or remember what function the code at the label START performs. A better comment would be:

```
JMP START    ;Get the next character.
```

or whatever the reason may be. Documentation is said to be four times the cost of actually writing the program code. In practice I have seen the ratio climb as high as 10:1 due to the cost involved in writing program documentation months after the program was written.

Even worse is when you encounter a program listing which runs on page after page after page, without a single comment. Woe to the poor soul whose job it is to patch or understand that program. We're all guilty of not documenting our programs in a textbook manner (Listing 2-1), yet the need to properly document your programs cannot be overestimated.

Your First AL Program

Let's jump right into programming in Assembly Language, and I'll explain the design process as we proceed. To enter the source program I'm about to describe, you'll need to use EDLIN or some type of word processing software. Once the source file has been created, you'll assemble, link, and debug the program.

Don't worry about not understanding what the source statements mean at this point. After I discuss the 8088 instruction set (Chapter 4) and MS-DOS functions in Chapter 6, you'll better understand the source statements found in the program.

The First Step: Defining the Problem

Every software project consists of several phases. A programmer I once worked with commented after a staff meeting that there were only three major phases to any commercial software project. The first phase is when the marketing department commits to a customer's schedule. The second phase is when the software department tells the marketing department they're nuts. And the third phase is getting the job done. I'll discuss only what's required to get the job done.

In reality, the first phase of any software project is to define what the program is attempting to solve. You can do this by means of an outline, using plain English statements to define the problem or by use of a program flow chart or state transition diagrams.

Figure 2-5 depicts the flow charting symbols used to describe program flow pictorially. Let's use these symbols as a design tool to illustrate our program's logic after we state what we want to do.

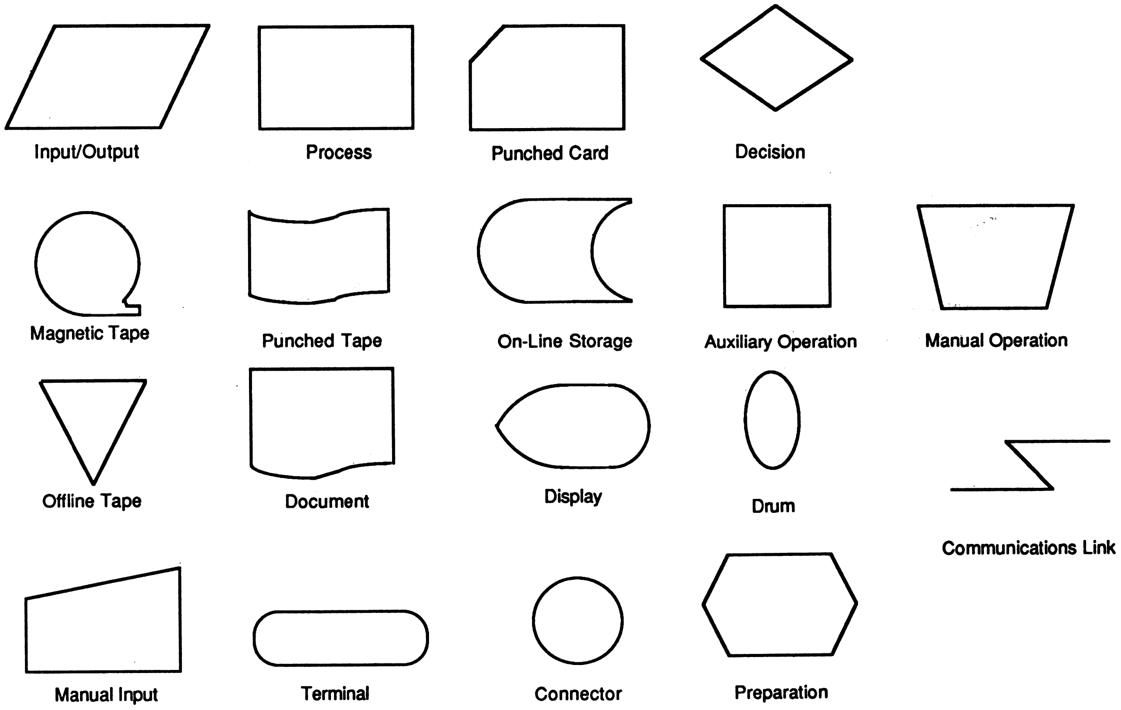

Input/Output
Process
Punched Card
Decision
Magnetic Tape
Punched Tape
On-Line Storage
Auxiliary Operation
Manual Operation
Offline Tape
Document
Display
Drum
Communications Link
Manual Input
Terminal
Connector
Preparation

Figure 2-5 Programmer's Flow Chart Symbols

Objective: Use system calls to read the system configuration of the computer the program is being executed on. Then display this information on the user's console.

It sounds simple enough. But how do we go about solving the problem? Put down some details as to exactly what we want to know about the system configuration.

I. How many disk drives?
II. The system's memory size.
III. The number of printers available?
IV. Communications Port?
V. Graphics board?
VI. Game Port Attached?

Create a flow chart of the program and describe each process as it relates to the diagram. Figure 2-6 is the flow diagram I designed. You may have come up with a different flow chart, which is perfectly legal. It's the results that count.

The Second Step: Writing the Source Code

This step is fairly straightforward. If you have created a good flow chart in the previous step, all that has to be done is to choose the proper 8088/86 instructions to accomplish the task. Listing 2-1 is the source code I used to solve the problem. You may, after becoming familiar with the instruction set of the 8088 and MS-DOS, have chosen a similar algorithm (solution) and similar instructions. You might also have chosen a totally different algorithm, or set of instructions, to perform the same task.

You'll enter the source code using EDLIN (see your *IBM DOS Operations Manual*) or another editor (the source code is also on the disk which accompanies this book). Use the filename CONFIGSY.ASM when you open the source file. Once the code has been entered, save the file to disk.

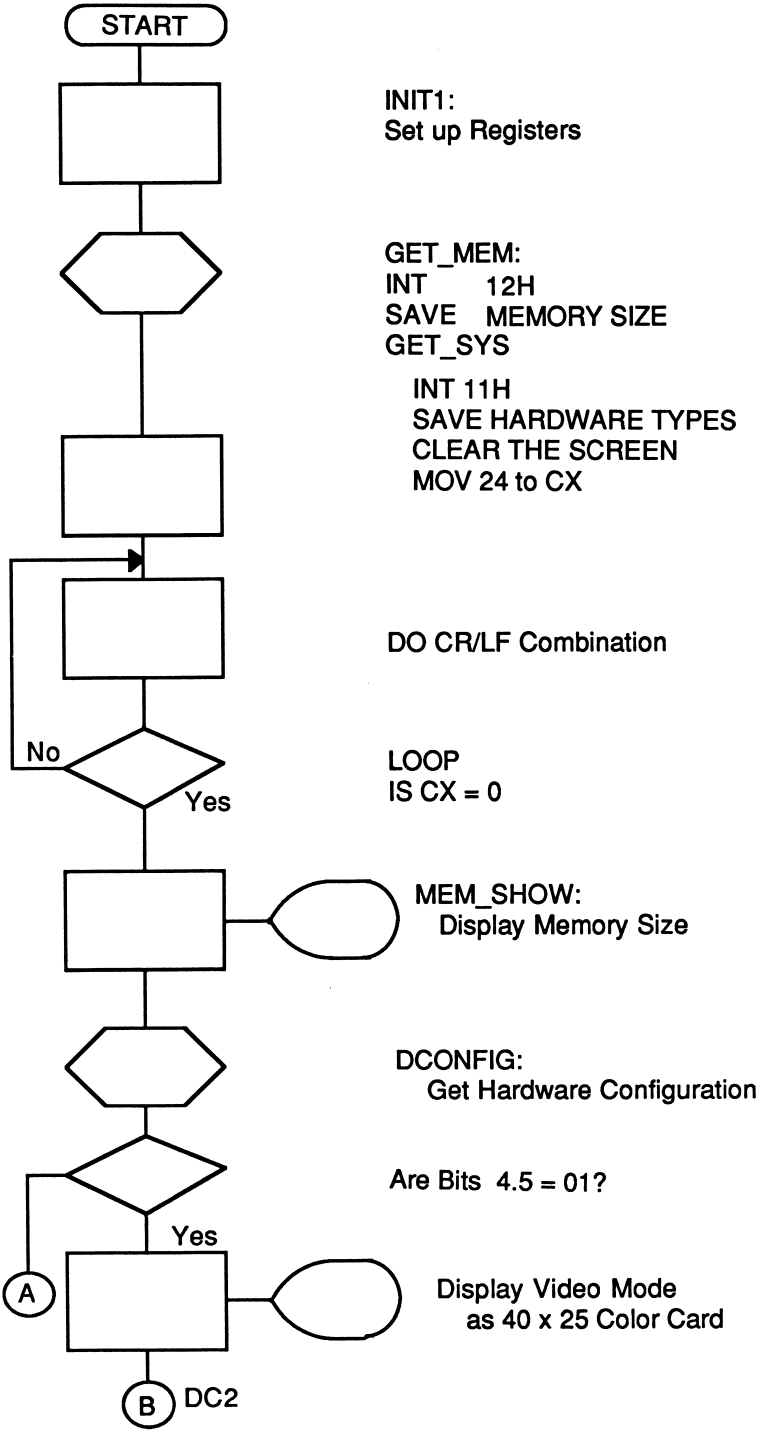

START
INIT1:
Set up Registers
GET_MEM:
INT 12H
SAVE MEMORY SIZE
GET_SYS
INT 11H
SAVE HARDWARE TYPES
CLEAR THE SCREEN
MOV 24 to CX
DO CR/LF Combination
No
Yes
LOOP
IS CX = 0
MEM_SHOW:
Display Memory Size
DCONFIG:
Get Hardware Configuration
Are Bits 4.5 = 01?
Yes
A
Display Video Mode
as 40 x 25 Color Card
B
DC2

Figure 2-6A

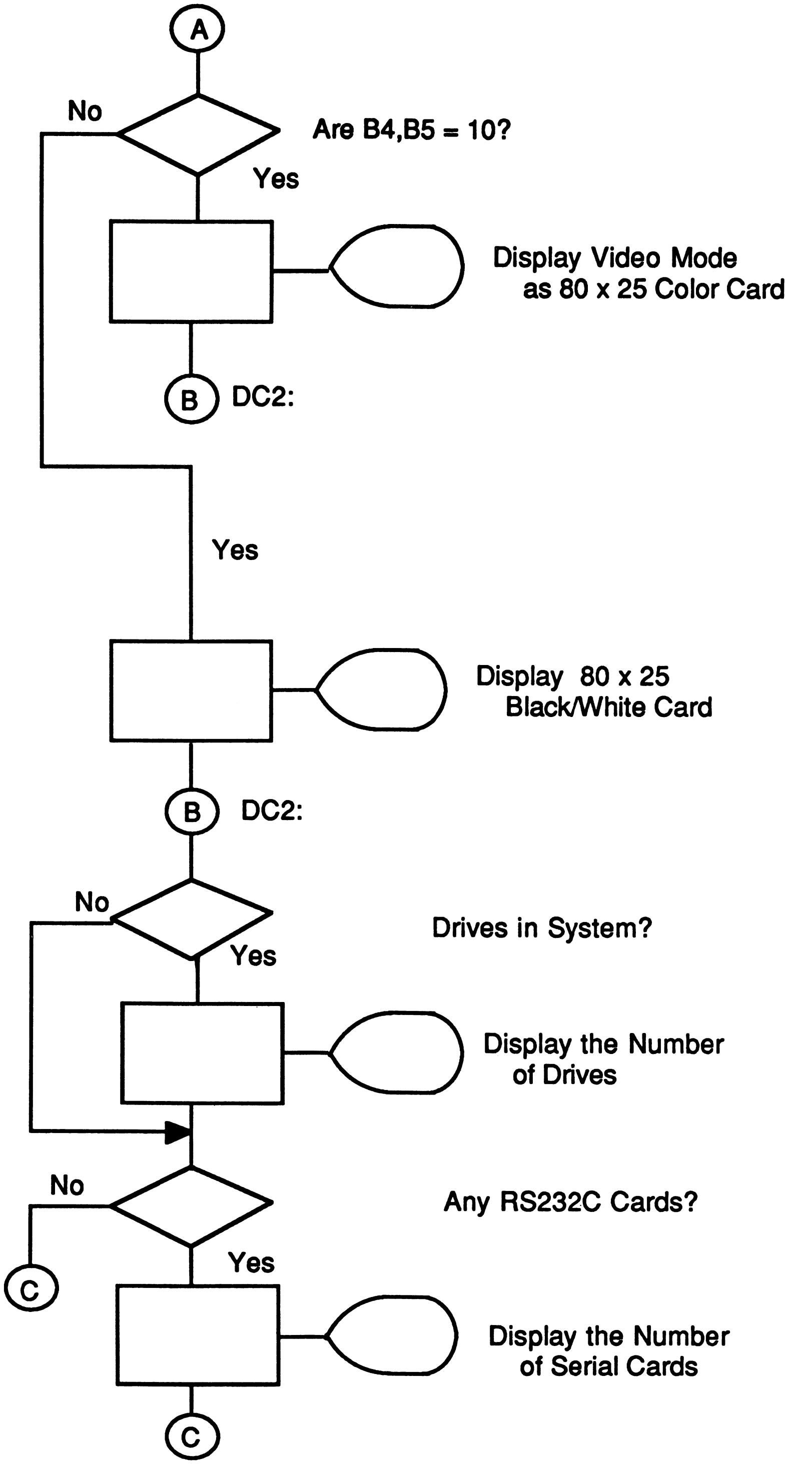

A
No
Are B4,B5 = 10?
Yes
Display Video Mode as 80 x 25 Color Card
B
DC2:
Yes
Display 80 x 25 Black/White Card
B
DC2:
No
Drives in System?
Yes
Display the Number of Drives
No
Any RS232C Cards?
Yes
C
Display the Number of Serial Cards
C

Figure 2-6B

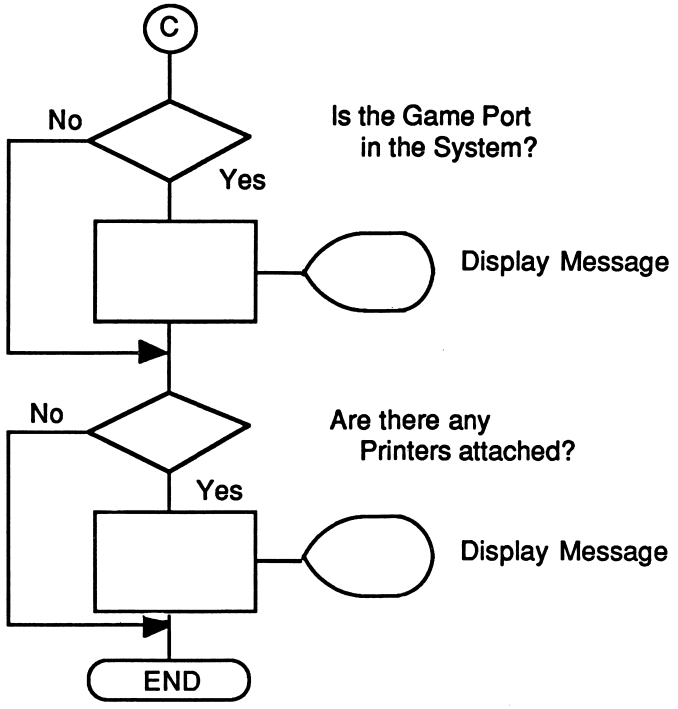

C
No
Is the Game Port
in the System?
Yes
Display Message
No
Are there any
Printers attached?
Yes
Display Message
END

Figure 2-6C

The Third Step: Using the Assembler

Once the source code has been entered and the file has been saved to disk, you'll invoke the assembler to translate the source program to machine language. From the MS-DOS prompt, type (Note: <CR> means press the return key):

```
MS DOS PROMPT                  You Type

A>                            masm <CR>

Source Filename [.ASM]:       CONFIGSY.ASM <CR>

Object Filename [.OBJ]:       CONFIGSY.OBJ <CR>

Source Listing [NUL.LST]:     CONFIGSY.LST <CR>

Cross reference [NUL.CRF]:    CONFIGSY.CRF <CR>
```

If there is an error, use the TYPE command from DOS to list your CONFIGSY.LST file. This file will contain any error messages and the line where they occurred. Compare the listing to the one shown in Listing 2-1, correct the error, and repeat the assembly process until the file assembles properly.

The Fourth Step: Using the Linker

Once the source file has been assembled, use the linker to produce an executable program file. From DOS type:

```
MS DOS                          You Type

A>                             Link

Object Modules [.OBJ]          CONFIGSY.OBJ <CR>

Run File [A:CONFIGSY.EXE]      <CR>

List File [NUL.MAP]            CONFIGSY.MAP <CR>

Libraries [.LIB]               <CR>
```

You have now created a file on disk, CONFIGSY.EXE, which can be used to find the configuration of your system.

The Fifth Step: Create a Cross Reference File

```
MS DOS                          You Type

A>                             CREF.EXE

Cross Reference [.CRF]         CONFIGSY.CRF

Listing Cross Reference.REF]   CONFIGSY.REF
```

The Files You Have Created

When you invoked the IBM Macro Assembler, you used your source file CONFIGSY.ASM as the input file to the assembler. You then specified the names of three files that were created by the assembler. CONFIGSY.OBJ is the object code output file. CONFIGSY.LST is the program listing that contains, on a line-by-line basis, the object code generated from translating each line of the source (.ASM) file.

The Object File

The object file, CONFIGSY.OBJ, contains information used by the linker to produce the executable file, CONFIGSY.EXE. The object file cannot be run on the computer; whereas, the executable file can be run by typing CONFIGSY.EXE from the MS-DOS command prompt. The linker creates an executable file by appending the necessary information MS-DOS requires to load and execute the program.

The Listing File

Listing 2-2 in Appendix D is the printout of the listing file, CONFIGSY.LST. The format of the listing consists of the following fields: line numbers, relative segment offsets, object code, and source fields.

Notice that the listing produced by the assembler contains line numbers in the left-most field. Line numbers and comments contained in the source file are not translated by the assembler; they produce no object code.

Assembler directives, such as the PAGE and TITLE directives, also produce no object code. Directives control and inform the assembler how to carry out the assembly process. For example, the PAGE directive commands the assembler to assemble the program formatted for a certain number of lines per page (56, in this example) containing a certain number of characters per line (132, in this example).

I've used many messages and defined them in the data segment. The object code produced by these source statements happens to produce ASCII code in the object file. Just as the programmer makes the decision to use a specific bit pattern to represent a signed or unsigned numeric value in binary, so is a decision made as to whether or not the bits contained in a data byte are to be interpreted as ASCII.

Look now at the code produced by MEM_MESS. 4DH is the ASCII representation of the letter M. 65H is the ASCII code for a lowercase e. Each letter of the message is translated to its ASCII equivalent.
The relative offset of the object code produced by each source statement (relative to the start of the segment) is contained in the second field. MEM_MESS begins at the offset 0AH from the start of the data segment. That is to say, the ASCII code for the letter M of the message block is stored at the tenth byte from the start of the segment.

Cross Reference File

The cross-reference file generates a detailed listing of all the symbolic names used in the program. Next to the label's name is the line number where the label originates, and every line number where the symbol name is referenced. For example, the line symbol MEM_MESS originates at line 25 and is referenced on line 128. To verify this, look at Listing 2-2 to confirm that the cross-reference listing is indeed accurate.

To generate the cross-reference listing, you must specify a cross-reference file name when running the assembler. You did this by specifying the file name CONFIGSY.CRF at the appropriate prompt. Next, you ran the cross-reference utility CREF.EXE, which used the CONFIGSY.CRF file as the input file and, in turn, produced a file named CONFIGSY.REF.

This information is extremely useful when you have to debug a program. You'll come to appreciate the many different types of listings the assembler and linker create when you have to debug your first program. It's a little like trying to read and understand a book in the dark. Debugging a program without the aid of a listing is just as impossible.

Chapter 3 describes the 8088's architecture. In subsequent chapters I'll use programs similar to this one to illustrate advanced features of the assembler.

Chapter 2 Review

1. You must obtain a PH.D. in applied physics before you can program in assembly language (True or False).
2. BASIC executes faster than all known language types (True or False).
3. To enter a source file, you must use an editor such as MASM (True or False).
4. Real programmers enter all their programs in machine language (True or False).
5. If you answered false to question four, why don't they?

6. The primary advantage assemblers have over writing programs directly in machine language is that you can use ___________ rather than ___________ addresses.
7. ___________ programming allows programs to be developed in smaller chunks, which makes the program easier to ___________, ___________ and ___________.
8. Each field in a source program must be separated by at least one ___________.
9. A well-documented program contains many ___________.
10. Why are comments important?

3

8088 Architecture

With the introductory chapters behind us, it's time to focus on the 8088 and the IBM PC. In this chapter I will examine the 8088's architecture and the architecture of the IBM PC. The term architecture, refers to what's inside the 8088 and the PC, from the programmer's point of view. In terms of the 8088, it's the register set, instruction set (discussed in Chapter 4), and the memory addressing modes available.

Microprocessors and Microcomputers

Computer architecture is composed of the three primary sections. Figure 3-1 illustrates the three main sections of a computer system: memory, input/output (I/O), and the central processing unit (CPU). The CPU contains an arithmetic logic unit (ALU) that carries out arithmetic operations for the CPU and storage locations called registers. The registers are used to move data to and from the CPU to memory and the I/O sections.

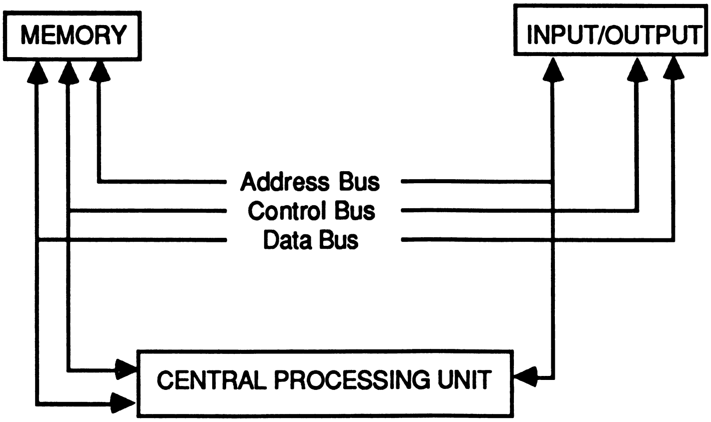


Figure 3-1 Sections of a Micro Computer

The control section in a microprocessor-based system is the brain of the system; it is the microprocessor itself, the CPU. Its function is to control data transfers to and from the microprocessor and the other sections of the computer. Most CPUs will sequentially fetch, decode, and execute the machine instructions as they come from memory.

The 8088 has a slightly different architecture, which speeds up this fetch/-decode/execute cycle. This is shown in Figure 3-2.

As can be seen, the instruction fetch and execution cycles of the 8088 are overlapped, allowing the 8088 to substantially increase the microprocessor's throughput. The 8088 implements two independent units which are internal to the chip. These are the BIU (Bus Interface Unit) and the EU (Execution Unit). The BIU of the 8088 fetches and reads and writes data via the system's bus. It fetches instructions from memory and places them in a 4-byte queue (6 bytes for the 8086). (Source Intel).

The EU removes the data and instructions placed in the queue by the BIU and executes them. This technique, which is referred to as pipelining, allows the fetch and execution cycles to overlap. While the EU is executing instruction, the BIU is fetching another from memory. Whenever the program branches to another part of the program, the queue is emptied and filled with data and instructions beginning at the new execution address.

The CPU also contains some general purpose and dedicated storage locations internally. These storage locations are referred to as registers. Usually a CPU will have at its disposal one or more accumulators that are used for arithmetic operations and one or more index registers that are used to hold addresses or pointers into memory. Additionally, there will be other registers designed with more specific uses in mind.

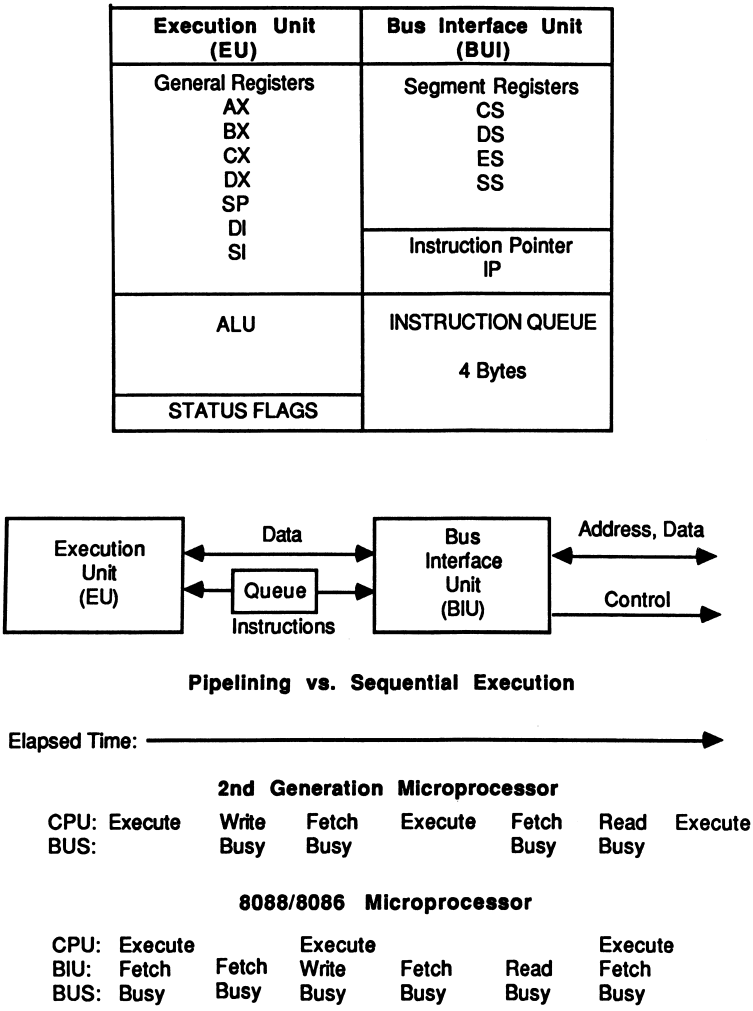

Execution Unit (EU)
Bus Interface Unit (BUI)
General Registers
AX
BX
CX
DX
SP
DI
SI
Segment Registers
CS
DS
ES
SS
Instruction Pointer
IP
ALU
INSTRUCTION QUEUE
4 Bytes
STATUS FLAGS
Execution Unit (EU)
Data
Queue
Instructions
Bus Interface Unit (BIU)
Address, Data
Control
Pipelining vs. Sequential Execution
Elapsed Time:
2nd Generation Microprocessor
CPU: Execute Write Fetch Execute Fetch Read Execute
BUS: Busy Busy Busy Busy
8088/8086 Microprocessor
CPU: Execute Execute Execute
BIU: Fetch Fetch Write Fetch Read Fetch
BUS: Busy Busy Busy Busy Busy Busy

Figure 3-2 8088 Architecture

Specialized Registers

The CPU's instruction pointer always points to the next instruction to be executed. The stack pointer points to the next available storage location in a RAM area known as the stack. The CPU has a register to record the status of the instructions the CPU executes. This register is referred to as the status, or flag, register.

Another important part of the CPU is the Arithmetic Logic Unit (ALU). Its purpose is to perform arithmetic calculations and set the flags in the status register accordingly. If one number is subtracted from another number and the result is zero, the ALU is responsible for setting the zero flag (ZF) in the status register. In the 8088, the ALU is a full 16 bits wide and can be used in 8-or 16-bit arithmetic operations.

Memory

The memory section stores the program instructions in the binary format discussed in Chapter 1. The memory section also holds data required by the program. The amount of binary data each memory location in the PC is capable of storing is one byte. An analogy can be drawn between the way these storage locations are accessed and the way a mailman delivers mail to someone's house. A unique address is assigned to each home on a given street. The same is true for each available storage location in memory.

The CPU is the microprocessor's mailman. It generates the desired address by placing the proper electrical signals at the physical pins of the processor chip designated for this purpose. The memory section responds by allowing the CPU to read data (accept mail) from the specified memory location or by allowing the CPU to write data (deliver the mail) into the location being accessed. Addresses start at memory location zero and continue to the maximum address the microprocessor can generate. In the case of the 8088, this upper limit is FFFFFH, which is over one million memory locations (1,048,576, to be exact). This corresponds to the total possible binary combinations the 8088 can generate via its 20 address lines (2 raised to the 20th power = 1,048,576).

RAM, an acronym for Random Access Memory, is a type of memory that can be read from and written to. The programs you write for the IBM PC are read from disk and placed in RAM for execution. Unfortunately, whenever the computer's power is turned off, the information stored in RAM is lost.

In fact, the RAM in the IBM PC (and most computers) needs to be refreshed periodically while the system is on. This type of memory is known as Dynamic Random Access Memory, or DRAM. Each bit of memory can be thought of as small capacitor which stores a charge. Without periodically refreshing each bit cell, the cell's charge will slowly discharge. Memory refresh is nothing more than the system reading the contents of a given RAM location and writing that value back to the same location, thereby replenishing the charge for that bit. Refresh is transparent to the user and is something the Assembly Language programmer normally is not concerned with.

ROM (Read Only Memory) can only be read from, not written to. The data contained within the ROM is therefore said to be unalterable or nonvolatile. ROM is used in the IBM PC to store the program that monitors the system's resources when the computer is turned on. The program checks to see what I/O devices are attached to the system, how much RAM is installed in the system, and if the resources are functional. The ROM also contains the BASIC interpreter and individual routines that form what is known as the computer's Basic Input/Output System or BIOS. These routines allow a convenient method for the applications programmer to access the system's resources: printers, disks, keyboard, video, etc.

A small program called a *bootstrap* loader is also contained in the ROM. Its function is to read in the first sector of information from the diskette containing MS-DOS. Once this information is read from disk into the system's memory, the ROM passes control to the program read from disk, which loads the rest of DOS into memory.

Segmentation

The 8088 views its 1M byte of memory as being comprised of segments that can be up to 64K bytes in length. There is a memory segment for your code, data, and stack. Each segment is associated with specific registers, known as code segment registers, and pointers that contain offsets into the segment. By using the segment and offset registers, data may be accessed anywhere within the segment. I'll discuss the 8088's segment registers and memory segmentation in detail later in this chapter.

How Data Is Stored in Memory

The 8088 microprocessor stores 16-bit values in memory in what appears to be reverse order. The low order byte of a 16-bit value is stored in the lower memory location, with the high order byte being stored in the next higher memory location. In a moment I'll discuss the various registers within the 8088. When we visualize how data is stored in the registers and contrast this view with how data is stored in memory, it appears as if it has been saved in reverse. Figure 3-3 demonstrates how the data value 0F07H is stored in the 8088 register AX and how the value would be stored in memory.

As can be seen from Figure 3-3, values stored in memory appear to be stored in reverse when compared to the same value as it is stored in a register. Remember, AX is comprised of two 8-bit registers, AH and AL. AH is the high order 8-bit register, and AL is the low order 8-bit register. Therefore, when the entire 16-bit word is stored in memory, AL is stored at the lower memory location, and AH is stored in the higher memory location. When data is read from memory, the low order memory byte is placed in AL, and the high order byte is placed into AH.

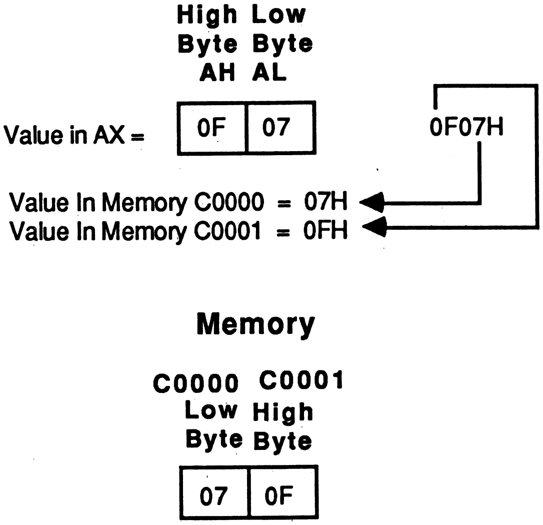


Figure 3-3 Data Storage in Registers and Memory

Procedures and Stacks

Procedures, or subroutines, are specialized and often-used mini-programs. The functions they perform may be required in several places in a large program. Rather than writing the same instructions over and over again, the routine is made into a procedure that is called as needed by the main program. This helps to conserve memory and makes the program easier to maintain should the procedure ever need to be rewritten.

RAM is also used for temporary storage of data in what is referred to as the STACK. Stacks are areas in RAM that are used by the microprocessor to save certain registers during special operations.

In much the same manner as memos are deposited and removed from an IN basket on an office desk, so are addresses and data pushed and popped from the stack. The last piece of paper put into the IN basket is the first one taken from basket. In the 8088, a register known as the stack pointer (SP) points to the top of the stack (TOS). When values are pushed on the stack, the stack pointer is decremented by 2, and the word value is written to the memory pointed to by the stack pointer. Each push stores 2 bytes (a word) of information on the stack. Figure 3-4 illustrates how the stack grows downward in memory each time data is pushed onto it.

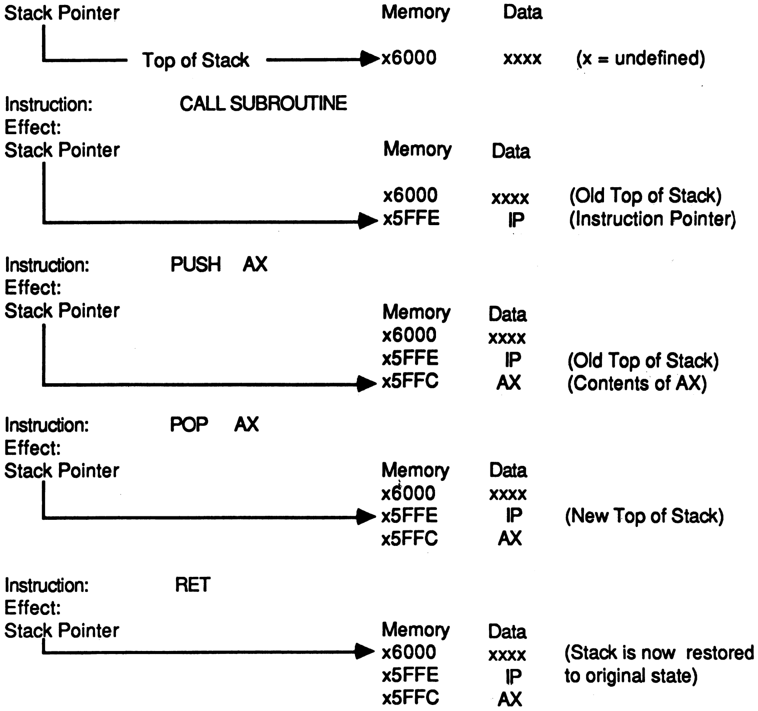

Stack Pointer
Memory
Data
Top of Stack
x6000
xxxx
(x = undefined)
Instruction:
CALL SUBROUTINE
Effect:
Stack Pointer
Memory
Data
x6000
xxxx
(Old Top of Stack)
x5FFE
IP
(Instruction Pointer)
Instruction:
PUSH AX
Effect:
Stack Pointer
Memory
Data
x6000
xxxx
x5FFE
IP
(Old Top of Stack)
x5FFC
AX
(Contents of AX)
Instruction:
POP AX
Effect:
Stack Pointer
Memory
Data
x6000
xxxx
x5FFE
IP
(New Top of Stack)
x5FFC
AX
Instruction:
RET
Effect:
Stack Pointer
Memory
Data
x6000
xxxx
(Stack is now restored to original state)
x5FFE
IP
x5FFC
AX

Figure 3-4 Stack Operation

When a procedure is called from the main program, the processor saves the address of the instruction immediately following the CALL instruction. This pointer is saved by the 8088 by pushing it onto the stack. The address of the procedure that was called now becomes the new instruction pointer.

The last instruction executed in the procedure is the return instruction, RET. When the RET instruction is encountered, the value previously pushed onto the stack is retrieved (popped off the stack), replacing the contents of the instruction pointer (IP). Control is returned to the instruction immediately following the instruction that called the procedure.

When a program calls a procedure which resides in another segment, not only is the value of IP saved on the stack, but so are the contents of the segment register CS (code segment). CS is then loaded with the segment address of the procedure, and IP is loaded with the offset of the first instruction in the called procedure. On return, the previously saved register values are popped from the stack, allowing program execution to resume with the instruction immediately following the call instruction.

When data are popped from the stack, the word value stored at the top of the stack is written to the register designated in the POP instruction, and the stack pointer is incremented by 2 to point to the new top of stack.

As Figure 3-4 shows, the CALL instruction decrements the stack pointer by 2 and places the contents of the instruction pointer in the location pointed to by the pointer. The PUSH instruction also decrements the stack pointer by 2 and places the contents of the AX register (in our example) onto the stack. POP and RET each copy a word value from the top of the stack and increment the stack by 2. POP retrieves a value and places it into the register specified. RET always fetches either a word or double word from the stack and places it into either IP (intrasegment RET) or into CS and IP when an intersegment return is required. Notice how the stack grows downward in memory each time a value is pushed onto the stack and shrinks upward with each POP.

The stack is also a convenient place for the programmer to temporarily store data. As an example, perhaps you need to use the AX register for an operation, but the value currently in the AX register is of importance and cannot be lost. You need a quick and convenient place to store the current contents of AX. A commonly used technique is to save the register's contents on the stack, perform the required operation and restore the register with its original contents by popping its previous value from the stack.

I'll discuss interrupts in a moment, but for now, keep in mind that the CPU, on receipt of an interrupt, saves not only the instruction pointer on the stack, but the status flags and the code segment (CS) register as well. When the interrupt service routine is finished, an interrupt return (IRET) instruction is executed which restores the register's CS, IP, and the status register with the values previously stored on the stack.

Input/Output Section

The I/O section of a computer consists of the physical interface between the microprocessor and some type of peripheral device, such as a keyboard, printer, video display unit, or a disk drive.

In other applications, the microprocessor may be required to communicate with more exotic devices, such as an analog-to-digital converter, which is designed to translate an analog voltage level to a binary word the computer understands. The A/D may be attached to a sensor or a transducer and incorporated in machines used in a factory environment.

Whether the I/O device is a switch attached to the IBM or a transducer's output which is attached to an A/D expansion card in the IBM, some type of interface between them and the microprocessor is required. Without an I/O section, humans would never be able to communicate with the computer, nor would the computer be able to communicate with real world events.

Bus Architecture (Hardware)

The Control Bus

The CPU issues control signals over what is known as the control bus. These signals control data transfers between the CPU, memory, and the I/O section of the computer. The 8088 is capable of operating in two modes, minimum and maximum. The control signals generated by the 8088 are different in each of the two modes.

The signals generated and received by the 8088 form three main bus architectures within the system (Figure 3-5B). The address bus is composed of the signal lines AD0–AD7 and A8–A20. AD0–AD7 are time multiplexed, which means that at one particular instant they are used for transmitting an address to the memory or I/O sections, and at some other instant in time they are used for transmitting and receiving data. Therefore, AD0–AD7 comprise the data bus of the computer system. The control bus is comprised of all the other signals (with the exception of the VCC and GND signals, which are part of the power bus and supplies the 8088 with the necessary operating voltages.)

Minimum Mode

The control signals generated by the 8088 microprocessor in minimum mode are M*/IO, RD*, and WR* (* means that the signal must be a binary 0 or a logic 0 to be considered true).

M*/IO tells the system whether the operation is a memory transfer or one involving Input/Output (I/O) devices. If the line is low, a memory access is assumed. If the line is high (logic 1), an I/O operation is dictated. The RD* signal allows the CPU to read data from memory, and the WR* signal, to write data into memory. The CPU generates all the necessary bus control commands.

Maximum Mode

In maximum mode, the signals which were used in the minimum mode are

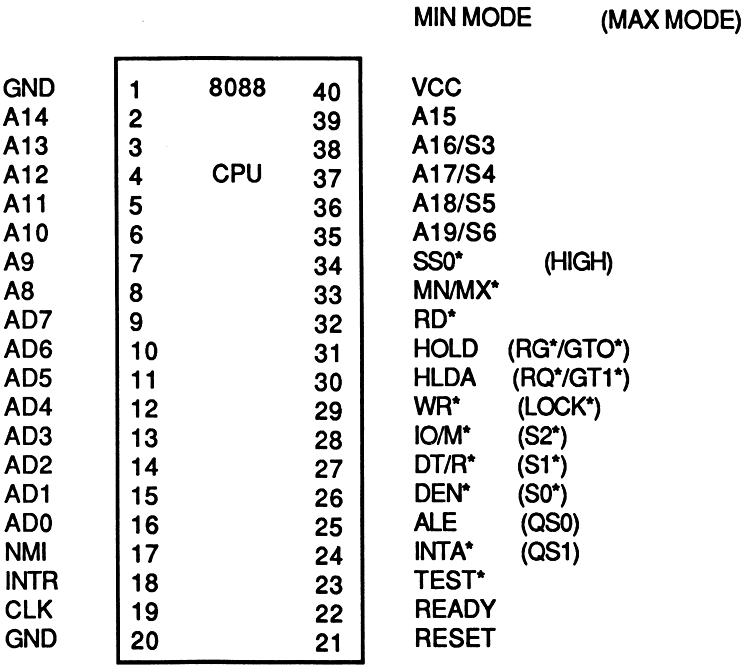

MIN MODE
(MAX MODE)
8088
CPU
GND 1 40 VCC
A14 2 39 A15
A13 3 38 A16/S3
A12 4 37 A17/S4
A11 5 36 A18/S5
A10 6 35 A19/S6
A9 7 34 SS0* (HIGH)
A8 8 33 MN/MX*
AD7 9 32 RD*
AD6 10 31 HOLD (RG*/GTO*)
AD5 11 30 HLDA (RQ*/GT1*)
AD4 12 29 WR* (LOCK*)
AD3 13 28 IO/M* (S2*)
AD2 14 27 DT/R* (S1*)
AD1 15 26 DEN* (S0*)
AD0 16 25 ALE (QS0)
NMI 17 24 INTA* (QS1)
INTR 18 23 TEST*
CLK 19 22 READY
GND 20 21 RESET

Figure 3-5A Bus Architectures

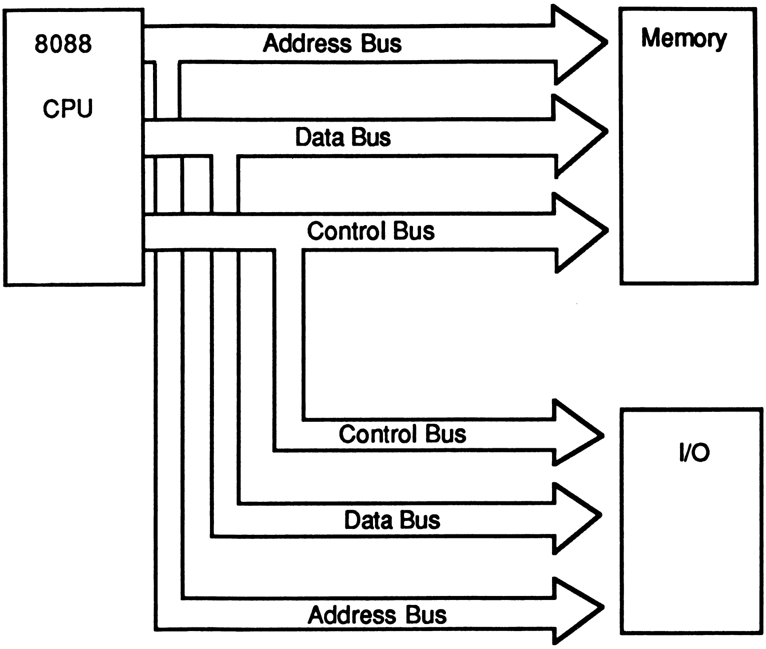

8088
CPU
Address Bus
Data Bus
Control Bus
Memory
Control Bus
Data Bus
Address Bus
I/O

Figure 3-5B Diagram of the Three Main Bus Architectures

redefined to support multiprocessor (more than one microprocessor) configurations. Fortunately for you, the configuration of the 8088 as used in the IBM PC is the maximum mode. This allows you to add co-processors like the Intel 8087 numeric processor, which speeds up mathematical calculations.

The Address Bus

In conjunction with the control bus, the 8088 issues the appropriate address for the operation specified. The 8088 is capable of addressing over 1M (million) bytes of memory via address lines A0–A19. The address lines are unidirectional; they are outputs from the 8088 and serve as inputs to the memory and I/O sections of the system.

The 8088 can generate up to 65,536 I/O port addresses by using the low order (A0–A16) address lines of the address bus. The 8088 is also capable of byte or word I/O, just as it is capable of performing byte or word memory transfers.

The Data Bus

The 8-bit data bus of the 8088 transfers data, 8 bits at a time to and from memory and the CPU. The data lines of the 8088 are shared with the address lines A0–A7. Signals S0, S1, and S2 are used to inform the 8288 bus controller of the type of bus cycle currently being executed (Figure 3-6).

When used in a maximum configuration, the 8088 supplies the 8288 bus controller with three signals, S0–S2. These signals are used to inform the 8288 of the type of bus cycle currently being executed. The 8288 generates the appropriate signals to demultiplex and latch the contents of the AD0–AD7 and execute the desired function (memory or I/O read or write, etc.). (Source Intel, used with permission).

The 8288 chip decodes the status lines and generates the necessary signals to control other support chips known as latches. Since the address lines A0–A7 may contain either data or address information, these lines are said to be multiplexed. The 8288 bus controller generates a signal DEN, which is used to enable data latches when the lines contain data. A signal ALE is used to enable address latches when the information on A0–A7 is to be interpreted as part of an address. Figure 3-7 depicts the possible states of the status lines S0–S2 and their interpretations.

A related processor manufactured by Intel is the 8086. The major difference between the 8088, which is the microprocessor in the IBM PC, and the 8086 is that the 8088 must always transfer data 8 bits at a time, while the 8086 is capable of transferring data 8 or 16 bits at a time. There are other minor differences between

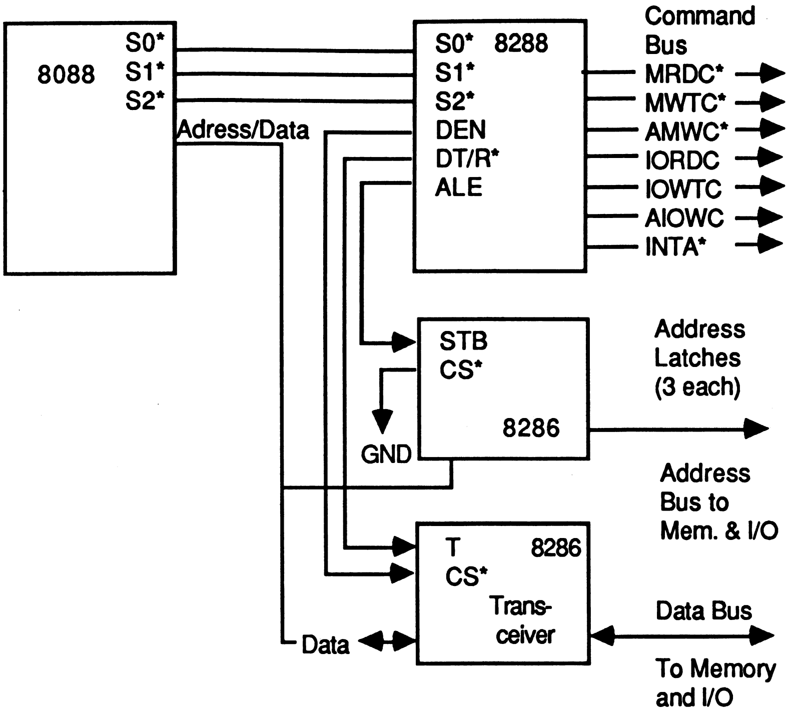

8088
S0*
S1*
S2*
Adress/Data
S0* 8288
S1*
S2*
DEN
DT/R*
ALE
Command
Bus
MRDC*
MWTC*
AMWC*
IORDC
IOWTC
AIOWC
INTA*
STB
CS*
8286
GND
Address
Latches
(3 each)
Address
Bus to
Mem. & I/O
T 8286
CS*
Trans-
ceiver
Data
Data Bus
To Memory
and I/O

Figure 3-6 8088 and 8288 Bus Controller (Maximum Mode)

S0	S1	S2	Meaning
0	0	0	Interrupt Acknowledge
0	0	1	I/O Read
0	1	0	I/O Write
0	1	1	HALT
1	0	0	Code Access
1	0	1	Read Memory
1	1	0	Write Memory
1	1	1	Passive

Figure 3-7 Status Line States

the 8088 and the 8086, but they can be considered insignificant from the programmer's point of view. Both microprocessors execute the same instruction set.

The primary difference between a microcomputer and microprocessor is that a microcomputer contains the three primary sections: CPU, memory, and I/O. A microcomputer can consist of a microprocessor and a separate memory and I/O section. This type of system architecture can be found in the PC. However, there are single chip microcomputers available as well. The Intel 8748, 8031, 8051, and 8096 are all microprocessors that incorporate the three sections of a microcomputer on a single chip. Therefore, they are referred to as a microcomputer chip and not as a microprocessor chip.

PC Memory Map

Figure 3-8 shows the memory map of the IBM PC. Notice that not all of memory is available to you. Some of the memory is dedicated to storing values that are required by the system to function properly. Many of these values serve as address pointers to service routines which the 8088 vectors to under certain conditions. Specifically, the vectors are used by the 8088 to find interrupt service routines, routines that are executed any time the 8088 is interrupted.

The 8088 Microprocessor

Internal Architecture

The 8088 contains an internal register architecture consisting of fourteen 16-bit registers. There are three primary classifications to the register set: data and pointer registers, segment registers, and control registers. The data registers AX, BX, CX, and DX can be used in 8-bit or 16-bit operations. If the X designation is used, as in AX, the reference implies 16 bits. However, you can also reference either the high order or low order byte of each data register by using the designations of H or L, as in AH or AL.

I) Data registers: AX, BX, CX, DX, SI, DI, BP, SP:	
Word length in bits	
b15..........b0	{b0-b7 = Low order half of register}
	{b8-b15 = High order half of register}
AX (Accumulator)	{AH and AL used in 8-bit operations}
BX (Base)	{BH and BL " " " " }
CX (Count)	{CH and CL " " " " }
DX (Data)	{DH and DL " " " " }
Index registers:	DI (Destination index)
	SI (Source index)
Pointer registers:	SP (Stack pointer)
	BP (Base pointer)
II) Code segment registers:	**DS, ES, CS, SS:**
	DS (Data segment)
	ES (Extra segment)
	CS (Code segment)
	SS (Stack segment)
III) Control registers: IP, SR:	
	IP (Instruction pointer)
	SR (Status register)

Segmentation

The 8088 has been shown to contain 20 physical address lines capable of addressing over 1M byte of memory. You'll notice, however, that the largest register within the 8088 holds only 16 bits (16 bits can generate a maximum of 64K addresses). How is it then, that we can access over 1 million bytes with the 8088? The answer is not at all obvious.

The processor calculates where in the one million bytes of memory each logical 64K segment resides by using a pointer in conjunction with a segment register. The segment registers, CS, DS, ES, and SS point to the beginning of specific segments. Each of the segment registers are associated with other registers that contain an offset into the 64K segment to facilitate the addressing of information within the segment.

Address	
00000	On Board System Ram 64K to 256K Maximum Note: Locations 00000 to 003FFH contain Interrupt Pointers to service routines
40000	Up to 384K Ram Expansion in I/O Channel
A0000	Reserved (128K) A4000 - C0000 Video Graphics, Display Buffer B0000 - B4000 Used for Monochrome Graphics B8000 - BC000 for Color/Graphics
C0000	ROM Expansion C8000 used for Fixed Disk control (192K Maximum)
F0000	Reserved (16K)
F4000 FFFFF	(48K Base System ROM) F4000 - F6000 (User ROM Area) F6000 - FE000 (Cassette Basic) FE000 - FFFFF (BIOS ROM)

Figure 3-8 IBM PC Memory Map

For example, the CS register holds the start address of the current 64K code segment. The instruction pointer, IP, contains an offset into the code segment, which points to the next instruction to be executed. The 8088 calculates the effective address by shifting the contents of the segment register left 4 bit positions and filling the LSBs with zeros. This in effect appends 4 bits of zeros to the value contained in the segment register. The segment offset contained in IP is added to the 20-bit value obtained by shifting the code segment register 4 bit positions. The result is the effective, or actual, address required by the operation.

Some examples of how a physical address is calculated follow:

```
Assume DS = B7A2H and SI = 001AH

              B7A20    <--- Shifted value of DS
          +    001A    <--- Offset into segment
          ---------
              B7A3A    <--- Effective address.

Assume CS = 46DDH and IP = A206H

Then -->      46DD0    <--- Shifted value of CS
          +    A206    <--- Offset into segment
          ---------
              50FD6 H <--- Effective address.

Assume SS = 70DFH and SP or BP = D209H

Then -->      70DF0    <--- Shifted value of CS
          +    D209    <--- Offset into segment
          ---------
              7DDF9 H <--- Effective address
```

Figure 3-9 illustrates the concept of segmented memory. Notice which pointer registers are associated with each segment register. In the 8088, instructions are always accessed with CS plus the offset of IP. Similarly, the stack is always accessed by using the SS and an offset contained in SP or BP. The data segment can be accessed by using either DS or ES and the offset specified in SI, DI, or BX. ES points to the extra segment and is associated with DI, while DS points to the data segment and is associated with SI. The extra segment and data segment are both used for data storage.

Segments can be organized in memory so that they are adjacent to each other. They may also be arranged in such a manner that they overlap, either partially or fully. When segments overlap, different programs, or tasks, can share data. When segments are arranged in an adjacent manner, programs can be written that allow data to be operated on locally by one task and not be shared by others. Segments must start on a 16-byte boundary in memory and must NOT be larger than 65,536 bytes.

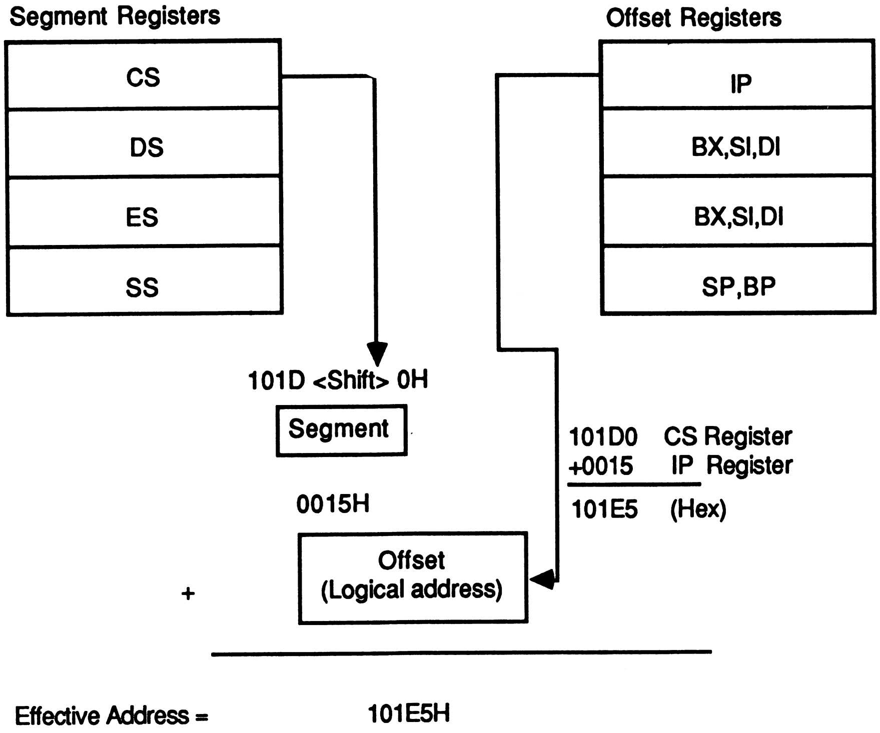

Segment Registers
CS
DS
ES
SS
Offset Registers
IP
BX,SI,DI
BX,SI,DI
SP,BP
101D <Shift> 0H
Segment
101D0 CS Register
+0015 IP Register
101E5 (Hex)
0015H
+
Offset
(Logical address)
Effective Address =
101E5H

Figure 3-9 Address Generation using Segment Registers + Offset

Data Registers

Another concept associated with the 8088 is that of asymmetric registers. Asymmetric registers are registers dedicated to certain functions. For example, the CX register must contain a value which is used as a counter when using certain instructions. Table 3-1 summarizes the types of operations associated with each of the 8088's registers.

Pointer and Index Registers

The pointer registers (BP and SP) and index registers (SI and DI) are used to hold pointers to memory. They can contain either an address offset or a displacement value which is used to access the operand. BP (base pointer) and SP (stack pointer) are used to access data within the stack segment. BP can be used to access data in

Table 3-1
Summary of Register Operations

Register	Operations
AX	Word Multiply/Divide Word I/O
AL	Byte Multiply/Divide/IO Translate, Decimal, Arithmetic
AH	Byte Multiply/Divide
BX	Translate
CX	String Operations
CL	Variable Shift and Rotate
DX	Word Multiply/Divide Indirect Input/Output
SP	Stack Operations
SI	String Operations
DI	String Operations

(Source: Intel, used with written permission)

segments other than the current stack segment by specifying a segment override prefix within the instruction. SI and DI are offsets into the current data segments. SI is associated with the current data segment (DS), while DI is associated with the current extra segment (ES).

Status Flags

The flag register provides you with the information necessary to make decisions based on a previously executed instruction. Assume for a moment that you want to execute one of two possible portions of code. The decision to execute one or the other is based on the value stored in the accumulator, AX. If the value is zero, you want to execute routine number 1. If the value is nonzero, you want to execute routine number 2.

If the result is zero, the ZF (Zero Flag) is set. Your program can then use the 8088 instruction JZ (Jump if Zero), to transfer control to routine number 1.

Carry Flag — (CF) — (b0)

The carry flag signifies that a carry out of the most significant bit has occurred from a previous addition of two numbers. For 8-bit addition, the carry is set when the addition causes a carry to be generated from bit 7. Similarly, the carry is set during 16-bit addition when there is a carry out of bit 15. Subtraction causing a borrow into the MSB also causes CF to be set. This can occur when two values are subtracted or compared, and the destination operand is smaller than the source operand.

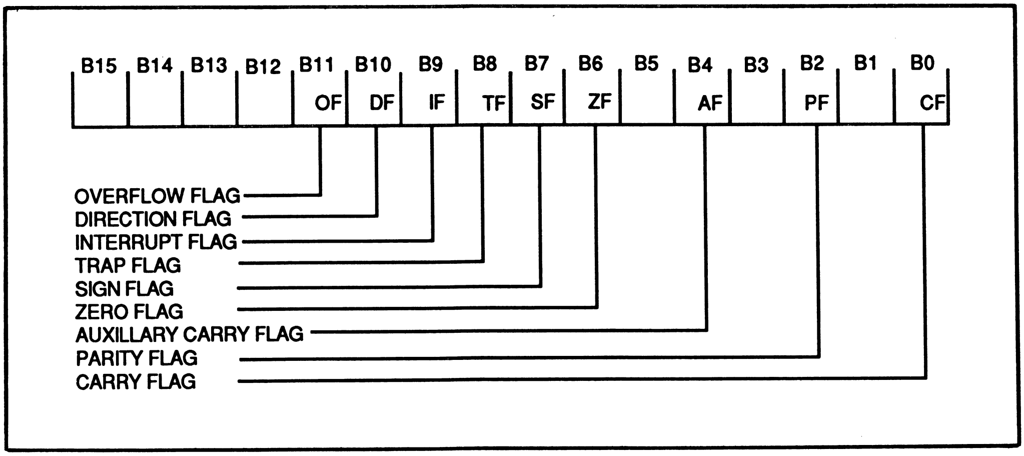

B15
B14
B13
B12
B11
B10
B9
B8
B7
B6
B5
B4
B3
B2
B1
B0
OF
DF
IF
TF
SF
ZF
AF
PF
CF
OVERFLOW FLAG
DIRECTION FLAG
INTERRUPT FLAG
TRAP FLAG
SIGN FLAG
ZERO FLAG
AUXILLARY CARRY FLAG
PARITY FLAG
CARRY FLAG

Figure 3-10 Status Register Flags

The carry can also be used to isolate a specific bit by rotating or shifting the bit into the carry flag. The program example at the end of Chapter 1 used this technique to determine if certain resources were available to the system. Bit 0 of the word used by MS-DOS to describe the system's equipment configuration is used to signify whether the system contains disk drives. By rotating b0 into the carry, a test can be made to determine whether the bit is set or reset.

Bit manipulations are not one of the strengths of the 8088 microprocessor, yet individual bits can be tested in the manner just described.

Parity — (PF)— (b2)

The parity flag is set if, after an operation, the total number of 1 bits contained in the result is even. If the total number of 1's is an odd number, the parity flag is reset.

Auxiliary Carry — (AF) — (b4)

The AF flag is set whenever a carry into (on subtraction) or out of (addition) bit 3 occurs. This can occur when adding, subtracting, or comparing numbers represented in packed binary coded decimal format. The 8088 sets this flag for any operation that causes a carry out of bit 3.

Zero Flag — (ZF) — (b6)

The zero flag is set if the result of an operation yields a zero result; otherwise, ZF is reset (0).

Sign Flag — (SF) — (b7)

The sign flag is set (logic 1) if the most significant bit of the result is set. If you recall from Chapter 1, signed numbers that are represented in binary notation use the MSB of a byte or a word as a sign bit. The bit is zero if the number is positive and 1 if the number is negative. SF is used to indicate whether the result is positive or negative and mirrors the MSB of the result.

Overflow Flag — (OF) — (b11)

The overflow flag is set whenever two signed numbers of the same sign (i.e., both positive or both negative) produce a result larger than the destination operand (the sign has changed). For example, when FFH is added to 80H, a result of 7FH is returned. Bit 7 in both operands is set, as both operands are negative values, but bit 7 in the result is a binary zero. The result of the addition causes the sign bit to change. This operation would cause an arithmetic overflow (OF = 1) and generate a carry (CF = 1).

Multiplication Indicators (CF and OF)

The carry and overflow flags also provide information about a previous word or byte multiplication. Multiplication of two 8-bit operands returns the high order half of the 16-bit result in AH and the low order half of the result in AL. If the operands are 16 bits in length, DX will contain the high order half of the 32-bit result and AX will contain the low order half. In either case, CF and OF are both set if the high order half of the result is a nonzero value for unsigned multiplication. When signed numbers are multiplied, CF and OF are both set if the high order half of the result is not a sign extension of the result; otherwise, CF and OF are both cleared.

Trap Flag — (TF) — (b8)

When the trap flag is set, the 8088 enters a single step mode of operation. This mode is quite useful for debugging a program. You can set or reset this flag. When this bit is set, the 8088 executes a type 1 interrupt after executing the current instruction. The debugging program's address must be specified in the type 1 interrupt vector (see Figure 3-8).

Interrupt Flag — (IF) — (b9)

The interrupt flag is used to enable or disable external interrupts. You can set or reset IF. If IF is set to 1, then interrupts are enabled. When IF is zero, interrupts are disabled. One interrupt which cannot be disabled in this manner is the NMI or nonmaskable interrupt. The NMI cannot be masked or disabled by clearing the IF bit in the flag register the way other external interrupts can.

Direction Flag — (DF) — (b10)

The direction flag is used during string operations and can be set by the programmer. With the 8088 it is possible to point one of the index registers (SI) to the source string and the other index register (DI) to the destination string. Then by using one of the string instructions which I'll discuss in Chapter 4, you can compare or move one string to the other. The state of the direction flag automatically increments or decrements the index registers, SI and DI, during a string operation. When DF is set to a binary 1, the index registers are incremented by 1. If DF is binary zero, then the index registers are decremented each time the string instruction is repeated.

Interrupts

Have you ever been working at the office when the telephone rings? While talking on the telephone, someone walks up to your desk. You put the party on the telephone on hold and find out what the person who interrupted the conversation wants. Then while taking care of the interruption, your boss sticks his head in the door and tells you to report to his office immediately. Let's examine this scenario in more detail.

Whether you know it or not, you have responded to multiple interrupt sources. You put the party on the telephone on hold and serviced the interruption of the person who walked up to your desk. While servicing this interrupt, let's also imagine someone brings you the mail. This interrupt is of a lower priority than the one you are currently servicing, so you ignore it for the time being. When your boss tells you to report to his office, you cannot ignore the interruption, and you immediately service the boss's request for attention. Your boss has the highest priority in this scenario.

You finish taking care of whatever it was that your boss wanted, and you return to your coworker waiting patiently at your desk. "Let's see, where were we? . . . Oh yes, I remember . . . ." You retrieve the conversation's status at the time of the boss's interruption and continue from the point where the conversation left off. You now finish your conversation with your coworker and return to your telephone conversation, "Let's see, where was I . . . Oh yes, I remember . . . ." You continue your telephone conversation from the point in the conversation where you left off. You finish the conversation, say your good-byes, and hang up. Now you remember the mail has arrived. You look through the mail for anything of importance and then return to your normal routine, that which you were doing before all these interruptions occurred.

Microprocessor interrupts are not much different from what has just been described. The microprocessor can be interrupted from its normal programming by external events, just as if the doorbell or telephone were to interrupt you from your normal chores. Interrupts are said to be asynchronous; they can occur at any time without being synchronized to other system components, such as the system's clock. Characters typed from the keyboard are good examples of asynchronous interrupt, as the characters are not precisely timed. They occur at irregular intervals. When the 8088 receives a keyboard interrupt due to a key being pressed or released, the processor goes to the appropriate interrupt routine to determine what key was pressed.

Just like the human analogy, the 8088 services interrupts in the following manner:

1. The 8088 first finishes the current instruction being executed,
2. fetches the interrupt vector number from the 8259A programmable interrupt controller (PIC),
3. saves the status flag register,
4. clears the interrupt flag (IF) and the trap flag (TF),
5. and saves its place in the program by pushing the code segment register (CS) and then the offset register (IP) on the stack.

When the 8088 has finished executing the interrupt routine, it restores the status, segment, and offset registers, which were saved at the time of the interrupt, and returns to the main program where normal program execution resumes.

Maskable and Nonmaskable Interrupts (NMI)

Certain interrupts can be masked (ignored). When IF is cleared in the flag register, the 8088 ignores external interrupt requests. When IF is set, the 8088 responds to the interrupts. Nonmaskable interrupts cannot be masked. They always grab the 8088's attention. Typically, the NMI interrupt is used to signal a low voltage condition or power failure.

You can think of maskable interrupts as those which can be screened. You can have a receptionist or the company's telephone operator take messages for you while you are in a meeting. The messages will be there after you finish your meeting.

Nonmaskable interrupts are those that must be serviced no matter what. Even though you have told the receptionist to "Hold all my calls while I'm in this meeting," should there be an emergency or if the boss should call, you'll be interrupted. Obviously, some interruptions will always occur and require immediate service.

Addressing Modes

There are several data accessing methods available to you. The 8088 microprocessor allows any of the following modes to be used when accessing data. The general format of addressing data is:

```
OP-CODE    destination, source
```

Register Direct

Register direct addressing is used to manipulate data directly contained in two specific registers. For example:

```
ADD BX,CX     {Add the contents of CX to BX}

MOV AL,BL     {Move the 8-bit value in BL to AL}

MOV CX,AX     {Move the 16-bit value in AX to CX}
```

Immediate

Immediate addressing operates on a constant value, which is contained in the instruction itself:

```
MOVE AX,1000    {Moves a word of data (1000) to AX}

MOV AL, -1      {Moves the constant -1 into AL}

ADD AL,10       {Adds 10 to AL}

ADD AX, THOUS   {Adds the value of THOUS (1000) to AX}
```

Memory Direct

In memory direct addressing, the operand's address is specified in the instruction:

```
MOV AX,COUNTER    {Moves the contents of the memory location
                   COUNTER to AX}

MOV COUNTER,AX    {Move the contents of AX to the memory location
                   COUNTER}
```

Memory Direct with Index

This form of memory addressing accesses values stored in memory by specifying the start of the data structure (as in the direct addressing mode above) as an offset contained in DI or SI. For example:

```
MOV SI,02H           ;A displacement of 2

MOV AX,COUNTER [SI]  ;Moves the contents of the memory location
                     ;(COUNTER +2) into AX.
```

Register Indirect

In register indirect addressing, the BX, BP, SI, or DI register contains the address of an operand. Using the memory location COUNTER from the previous example, one of the registers is loaded with the address of the location COUNTER. This can be done in one of two ways:

```
1) LEA SI,COUNTER         {Loads the effective addres of COUNTER
                           into the SI register}

or;

2) MOV SI,OFFSET COUNTER {Loads the segment offset value of
                          COUNTER into SI}
```

Once the address has been loaded into the SI, DI, BP, or BX register, you can use the register to access data stored at that location. For example:

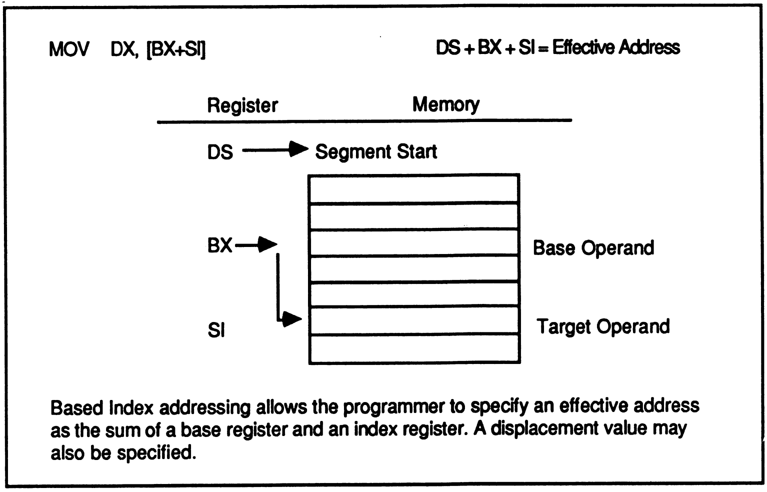
```
MOV AX,[SI]     {Moves the contents from the address pointed to by
                 SI into the AX register}

ADD AX,[SI]     {Adds the contents of the address pointed to by SI
                 to the AX register}

ADD [BX],AX     {Add the contents of AX to the memory location
                 pointed to by BX}
```

Based Index (Register Indirect with Index)

This mode of addressing accesses an operand in a data structure, pointed to by BX or BP, located at the offset specified by either DI or SI. For example, suppose DI contains the value 2, and BX contains the start address of a data structure that includes employees' hourly wages. The instruction MOV AX, [BX+DI] would load AX with a third employee's wages. Figure 3-11 illustrates register indirect with index addressing. Intel refers to this mode of addressing as based indexed addressing, because the effective address is calculated from the contents of one of the base registers and an index register.



Based Index addressing allows the programmer to specify an effective address as the sum of a base register and an index register. A displacement value may also be specified.

Figure 3-11 Register Indirect with Index

Register Indirect Indexed with Displacement Addressing

The last addressing mode is another variation of based index addressing. In register indirect with index plus displacement, the effective address is calculated as the sum of the contents of BX or BP, *plus* the contents of SI or DI, *plus* the specified displacement value. For example:

```
MOV AX, [BX+DI+2]      {Moves the contents of the memory location
                        pointed to by the sum of BX, DI, and 2 into
                        the AX register}

MOVE [BX+DI+2],CX      {Moves the contents of CX into the memory
                        location specified by the sum of BX + DI +2}
```

The displacement can also be specified in the form a symbolic name as: MOV MY__BYTE[BX] [DI] + 2,CX.

Memory-to-memory data transfers and manipulations are not allowed, nor are operations that specify a constant value as the destination operand.

Address Calculation Time

As you might have guessed, the calculation of an effective address takes a certain amount of time. The exact time required for address calculation depends on the addressing mode chosen. Table 3-2 can be used to determine effective address calculation times.

In Appendix B, you'll find the execution times of each 8088 instruction. The EA calculation time must be added to the instruction's execution time to determine the total execution time. For example, the NEG instruction lists an execution time for a memory operand as 16+EA. In all cases, four clock cycles must be added to the value shown (16, in this example), if the operation requires a 16-bit (word) transfer. The worst case, or longest execution time, for this instruction would then be 32 (16 + 4 + 12) clock cycles. The fastest execution time for this instruction would be when a register is specified as the operand. Since there is no effective address to be calculated, the instruction executes in three clock cycles.

Table 3-2

EA Components	**Clocks***	
Displacements Only	6	
Base or Index Only	5	(BX,BP,SI,DI)
Displacement + Base or Index	9	(BX,BP,SI,DI)
Base + Index	7	(BP+DI,BX+SI)
	8	(BP+SI,BX+DI)
Base + Index + Displacement	11	(BP+DI+DISP)
		(BP+SI+DISP)
	12	(BX+SI+DISP)
		(BX+DI+DISP)

*Add two clocks for segment override.
IBM system clock = 210 ns (nanoseconds)

Chapter 3 Review

1. There are ___________ main sections to a computer system.
2. The main sections are the ___________, ___________, and the ___________ section.
3. The ___________ section acts like a mailman to the other sections of a computer system.
4. The 8088 contains ___________ independent internal units.
5. The ___________ ___________ removes instructions from the ___________ and decodes and executes the instructions.
6. The 8088 flag register is ___________ bits wide.
7. The ALU of the 8088 is ___________ bits wide.
8. The maximum amount of memory the 8088 can directly address is ___________ bytes.
9. The Basic Input/Output System is contained in the system's

   A. RAM or

   B. ROM.

10. The 8088 views the total memory space as being divided into ___________ byte ___________.
11. An effective address is generated by taking the base value found in one of the ___________ registers and by adding an ___________ to that value.
12. Name at least two uses for the stack.
13. The 8088 is capable of generating ___________ I/O addresses.
14. The 8088 is operated in the ___________ mode in the IBM PC.
15. Name one difference between minimum and maximum mode of operation.
16. The data bus of the 8088 is ___________ bits wide.
17. The data bus of the 8086 is ___________ bits wide.
18. Low memory in the IBM PC is dedicated to storing ___________ vectors.
19. What is the difference between a microprocessor and a microcomputer?
20. There are ___________ 16-bit-wide registers in the 8088.

21. The data registers of the 8088 are __________, __________, __________, and __________.
22. There are __________ index registers.
23. There are __________ pointer registers.
24. There are __________ segment registers: __________, __________, __________, and __________.
25. The extra segment must always point to the stack (True or False).
26. __________ and __________ form a pointer to the next instruction to be executed.
27. The Z flag indicates that nothing is going on (True or False).
28. When the 8088 is interrupted, it will save the __________ register, __________ register and the __________ register on the stack.
29. Some interrupts may be __________. When this is done, the 8088 will/will not respond to external interrupts (circle the correct answer).
30. Name four addressing modes the 8088 is capable of employing.
31. An instruction's total execution time can be determined by looking up the instruction execution time in the __________, and adding the __________ calculation time to it.

4

8088 Instruction Set

We've covered a lot of ground in the first three chapters. Now it is time to tie the loose ends together (as far as the 8088 is concerned). We already know that the assembler translates source statements into the machine code the 8088 understands, but we have not described the 8088's instruction set, the commands which, when used other commands, create a program.

If you are an experienced assembly language programmer, you may be surprised to see some of the powerful instructions in the 8088's repertoire. There is much similarity between many high level languages and some of the instructions the 8088 is capable of executing. For example, you can implement functions similar to BASIC's FOR and NEXT iteration loops.

I have already used some of the 8088's instructions in previous chapters. By doing so, it is a bit like putting the proverbial cart before the horse. In order to illustrate an assembly language program as was done in the programming examples of Chapters 1 and 2, it was necessary to use the instructions which I am about to discuss. After reading this chapter, you should go back to the previous chapters and review the examples presented.

I also mentioned in Chapter 2 that an immediate value could not be used as a destination operand in an instruction. The reason is that it would make little sense to use immediate addressing with a destination operand of a constant. As the general format for a source code statement is:

```
OP-CODE destination, source
```

the destination operand would have to be capable of being altered. Immediate operands are constant values which are specified within the instruction. Therefore, they cannot be used as destination operands.

However, nearly all instructions allow you, the programmer, to specify memory or register operands as either source or destination operands. What if you want to add 20 to the contents of the memory location labeled COUNTER? Most processors first require that the value be fetched from memory and loaded into the accumulator. Then 20 would be added to the value in the accumulator and the new value saved in memory. The operation would require at least two instructions and perhaps three on most microprocessors. The 8088 accomplishes this operation in one instruction: ADD COUNTER,20. Allowing register and memory operands to be used as either source or destination operands in an instruction results in an overall reduction of the machine code generated by the assembler.

Instruction Groups

Table 4-1 shows the 8088 instruction set divided into six major groups.

Table 4-1
8088 Instruction Groups

Group 1
Data Transfer Instructions

General Purpose		Input/Output	
MOV	Move byte or word	IN	Input byte or word
PUSH	Push word onto stack	OUT	OUTPUT " " " " "
POP	Pop word off stack		
XCHG	Exchange byte or word		
XLAT	Translate byte		
Address Object		**Flag Transfer**	
LEA	Load effective address	LAHF	Load AH from flags
LDS	Load pointer using DS	SAHF	Save AH in flags
LES	Load pointer using ES	PUSHF	Push flags
		POPF	Pop flags

Group 2 Arithmetic Instructions	
Addition	
ADD	Add byte or word
ADC	Add byte or word with carry
INC	Increment byte or word by 1
AAA	ASCII adjust for addition
DAA	Decimal adjust for addition
Subtraction	
SUB	Subtract byte or word
SBB	Subtract byte or word with borrow
DEC	Decrement byte or word by 1
NEG	Negate byte or word
CMP	Compare byte or word
AAS	ASCII adjust for subtraction
DAS	Decimal adjust for subtraction
Multiplication	
MUL	Multiply byte or word unsigned
IMUL	Integer multiply byte or word
AAM	ASCII adjust for multiply
Division	
DIV	Divide byte or word (unsigned)
IDIV	Divide byte or word (signed)
AAD	ASCII adjust for division
CBW	Convert byte to word
CWD	Convert word to doubleword
Group 3 Logical, Shift, and Rotate	
Logicals	
NOT	Complement byte or word
AND	Perform logical AND on byte or word
OR	Perform logical OR on byte or word
XOR	Perform logical XOR on byte or word
TEST	Perform logical TEST (AND) on byte or word
Shifts	
SHL/SAL	Shift logical/arithmetic left byte or word
SHR	Shift logical right byte or word
SAR	Shift arithmetic right byte or word
Rotates	
ROL	Rotate left byte or word
ROR	Rotate right byte or word
RCL	Rotate through carry left, byte or word
RCR	Rotate through carry right, byte or word

continued

	Group 4 String Instructions	
REP	Repeat	
REPE/REPZ	Repeat while equal to zero	
REPNE/REPNZ	Repeat while not equal to zero	
MOVS	Move byte or word string	
MOVSB/MOVSW	Move byte or word string	
CMPS	Compare byte or word string	
SCAS	Scan byte or word string	
LODS	Load byte or word string	
STOS	Store byte or word string	
	Group 5 Program Transfer Instructions	
	Unconditional Transfer Instructions	
CALL	Call procedure	
RET	Return from procedure	
JMP	Jump	
	Conditional Transfer Instructions	
	Jump if	Flag Condition
JA/JNBE	Jump if above/not below	(CS or ZF)=0
JAE/JNB	Jump if above or equal/not below	CF=0
JB/JNAE	Jump if below/not above or equal	CF=1
JBE/JNA	Jump if below or equal/not above	(CF or ZF)=1
JC	Jump if carry	CF=1
JE/JZ	Jump if equal/zero	ZF=1
JG/JNLE	Jump if (<) / not > nor =	((SF xor OF) or ZF)=0
JGE/JNL	Jump if greater or =/not <	(SF xor OF)=0
JL/JNGE	Jump if less/not greater nor equal	(SF xor OF)=1
JLE/JNG	Jump if < or =/not >	((SF xor OF) or ZF)=1
JNC	Jump if not carry	CF=0
JNE/JNZ	Jump if not equal/not zero	ZF=0
JNO	Jump if not overflow	OF=0
JNP/JPO	Jump if not parity/parity odd	PF=0
JNS	Jump if not sign	SF=0
JO	Jump if overflow	OF=1
JP/JPE	Jump if parity/parity even	PF=1
JS	Jump if sign	SF=1
	Iteration Control Instructions	
LOOP	Loop	
LOOPE/LOOPZ	Loop if equal/zero	
LOOPNE/LOOPNZ	Loop if not equal/not zero	
JCXZ	Jump if register CX=0	
	Interrupts	
INT	Interrupt	
INTO	Interrupt if overflow	
IRET	Interrupt return	

Group 6 Processor Control Instructions	
Flag Operations	
STC	Set carry flag
CLC	Clear carry flag
CMC	Complement carry flag
STD	Set direction flag
CLD	Clear direction flag
STI	Set interrupt enable flag
CLI	Clear interrupt enable flag
External Synchronization	
HLT	Halt
WAIT	Wait
ESC	Escape
LOCK	Lock bus during next instruction
No Operation	
NOP	No Operation

Data Transfer Instructions

These instructions move data in and out of memory and the 8088's registers. They also move data contained in AL or AX to or from an I/O port. Intel further categorizes the data transfer instructions as:

A. general purpose data transfers,

B. I/O port transfers,

C. address-object transfers, and

D. flag register transfers

Arithmetic Instructions

Arithmetic instructions operate on signed or unsigned numeric operands in either binary or packed or unpacked BCD format.

Bit Manipulation Instructions

This group of instructions performs logical, shift, and rotate operations on byte or word operands.

Control and Transfer Instructions

The control and transfer instructions allow conditional or unconditional jumps to other parts of the program. Also included in this group are interrupt instructions and status flag operations.

String Manipulation Instructions
Possibly the most powerful of all the 8088's instruction types are those used for string manipulation. You can scan, move, or compare strings of data that are up to 64K bytes in length in these instructions.

Processor Control Instructions
This group of instructions allows you to set and clear the carry, direction, and interrupt flags in the 8088's status register. There are also instructions which synchronize the 8088 to other processors (referred to as co-processors) and external events.

Data Transfer Instructions

The first group of instructions consists of those that move data from one place to another in the PC. They move data either to or from:

1. Memory and registers,
2. One register and another register, or
3. The AL or AX register and an I/O port.

The data transfer instructions are the most fundamental and the most easily understood and are therefore the first ones I'll discuss. All of the instructions (for all groups) are listed alphabetically in Appendix C. They are reprinted in the appendix with the permission of Intel Corporation. The information which follows paraphrases the information contained in the appendix and expands upon it where necessary.

General Purpose Data Transfers

MOV

***Format:* MOV destination, source**

The MOV (move) instruction transfers either 8 bits (1 byte) or 16 bits (1 word) of data from the source to the destination operand. Any of the addressing modes discussed in Chapter 2 can be used to effect the transfer. This is the most frequently used 8088 instruction. Some examples of the MOV instruction are:

```
MOV AX,DX              ;Move the contents of DX to AX.
MOV BL,CL              ;Move the 8-bit contents of CL to BL
MOV AX,COUNTER         ;Move the contents of memory to AX
MOV AL,10H             ;Move the constant 16 (decimal) to AL.
MOV ES,AX              ;Move the contents of AX to the
                       ;Extra segment register.
```

The following forms of the MOV instruction are not allowed:

```
MOV 10H,AL                ;You cannot use a constant as the
                          ;destination.
MOV CS,AX                 ;You cannot use the code segment
                          ;register as the destination operand.
MOV ES,A000H              ;You cannot use a constant value
                          ;as the source operand and specify
                          ;a segment register as the destination.
MOV NEW_COUNT,COUNTER     ;You cannot use memory as both the
                          ;source and destination operands.
```

Let's examine each of the illegal MOVs in greater detail. I've already discussed why an immediate value cannot be used as the destination operand, but why can't we use the code segment register as the destination? Since the code segment register contains the base address of the program currently being executed and the IP register contains the offset into the code segment of the next instruction to be executed, altering CS produces a meaningless combination of the segment register and the instruction pointer. CS would be pointing to some new segment, while IP would still be pointing to the old offset of the old segment. Whatever IP would be pointing to in the new segment would not be what was desired; therefore, this form of the MOV instruction is not allowed.

The segment registers cannot be loaded from an immediate value in the source operand either. The assembler associates certain segment registers with the three types of segments in your program (code, data, and stack). In order to satisfy this assumption you must load the start address of the segment into one of the general purpose registers, and then transfer this value to the desired segment register. For example:

```
MOV AX,MY_DATA          ;Load the start address of the data

                        ;Segment into AX.

MOV DS,AX               ;Establish DS as pointer to MY_DATA
```

This is the method you should use to load the DS, ES, and SS registers with the starting address of a segment of memory. Remember, the segment registers shift this value left by 4 bit positions and append a low order nibble of zeros to the segment address (see Chapter 2).

Memory-to-memory transfers are also not allowed. A value must first be moved into one of the CPU's registers, and then written to the new location.

PUSH

Format: **PUSH source**

The PUSH instruction allows you to temporarily save data on the stack. This method of temporary data storage is often used when you want a register to perform some type of operation, but you do not want to destroy its current contents. The PUSH instruction can also be used to pass parameters (data) to subroutines or procedures. Once in the subroutine, the BP register is used to access the parameters that have been pushed onto the stack.

When the 8088 encounters the PUSH instruction, the stack pointer (SP) is decremented by two and the 16-bit source operand is transferred to the stack location pointed to by SP.

POP

Format: **POP destination**

The POP instruction is the complement of the PUSH instruction. It allows you to retrieve information that has been saved, or pushed, on the stack. When the 8088 encounters the POP instruction, the word pointed to by the stack pointer is transferred to the destination operand, which can be any 16-bit register (except CS) or a memory operand. SP is then incremented by two, and points to the new top of the stack.

A common mistake is neglecting to pop all the information which was pushed onto the stack. When a procedure is called by using the CALL instruction (which I'll discuss later in this chapter), the return address of the calling program is automatically saved on the stack by the 8088.

To exit the procedure and return to next the instruction in the program, the procedure must issue a RET, or return, instruction. When the 8088 encounters the RET instruction, the value pointed to by SP is transferred to the instruction pointer IP, assuming that both the main program and the procedure reside in the same code segment (intrasegment).

If the procedure and the calling program are not within the same segment (intersegment), the segment address as contained in the CS register is also saved on the stack. When the intersegment RET is encountered, the word pointed to by SP is popped off the stack into the instruction pointer. The stack is incremented by two, and the word pointed to by SP is popped from the stack into the code segment register. The stack pointer is again incremented by two to point to the new top of stack.

What would happen if you wanted to use AX in the procedure, but AX might contain a value used in the main program? Certainly you should save the value of AX prior to using the register in the procedure. Consider the following portion of code!

```
MAIN PROGRAM: CALL MY_PROCEDURE    ;Call a subroutine.

              MOV BX,AX            ;The procedure should return

                                   ;here.

              MOV AX,CX            ;etc.
```

The main program calls the procedure named MY_PROCEDURE. Assume for the moment that both the calling program and the procedure reside in the same code segment. The offset address into the current code segment is contained in IP and is automatically saved on the stack when the 8088 encounters the CALL instruction. A new offset value, which is the offset address of MY_PROCEDURE, becomes the new value of the instruction pointer IP. After the procedure is finished, control should return to the main program at the instruction which moves the contents of AX to BX.

Consider what would happen if the procedure were written in the following manner:

```
MY_PROCEDURE   PROC NEAR

               PUSH AX            ;Save tne value of

               . .                ;<The program statements of the
```

continued

```
                . .                     ;procedure. }
                RET                     ;Oh - Oh!
MY_PROCEDURE    ENDP
```

Here AX was saved on the stack and never restored the stack prior to exiting the procedure. When the RET instruction is executed, the 8088 transfers the value of AX that was pushed on the stack to the instruction pointer. Where do you think the 8088 will return to? Certainly not to the MOV instruction in the main program. It will try to execute whatever it finds at the memory location specified by the combination of CS and IP. Since the value in IP is whatever was in AX, chances are that the combination will not point to anything coherent.

The rule is: For every PUSH there must be a POP, and for every POP there must be a PUSH. The procedure should have been written as:

```
MY_PROCEDURE    PROC NEAR
                PUSH AX                 ;Save AX
                . .                     ;{Rest of the procedure}
                . .                     ;etc.
                POP AX                  ;Restore AX
                RET                     ;and return to caller.
MY_PROCEDURE    ENDP
```

You may want to save more than just one register when a procedure has been called. If this is the case, you must pop the words in the opposite order of that in which they were saved. Remember that the last value pushed on the stack is the first value to be popped.

```
MY_PROCEDURE      PROC NEAR
          PUSH AX                       ;Save AX, CX, and BX
          PUSH CX
          PUSH BX                       ;All registers have been saved
          . .                           ;Now execute procedure
          . .
          POP BX                        ;Restore the registers
          POP CX
```

```
        POP AX

        RET                 ;and return to caller.

MY_PROCEDURE ENDP
```

Figure 4-1 illustrates the stack usage due to the PUSH and POP instructions. (See also RET.)

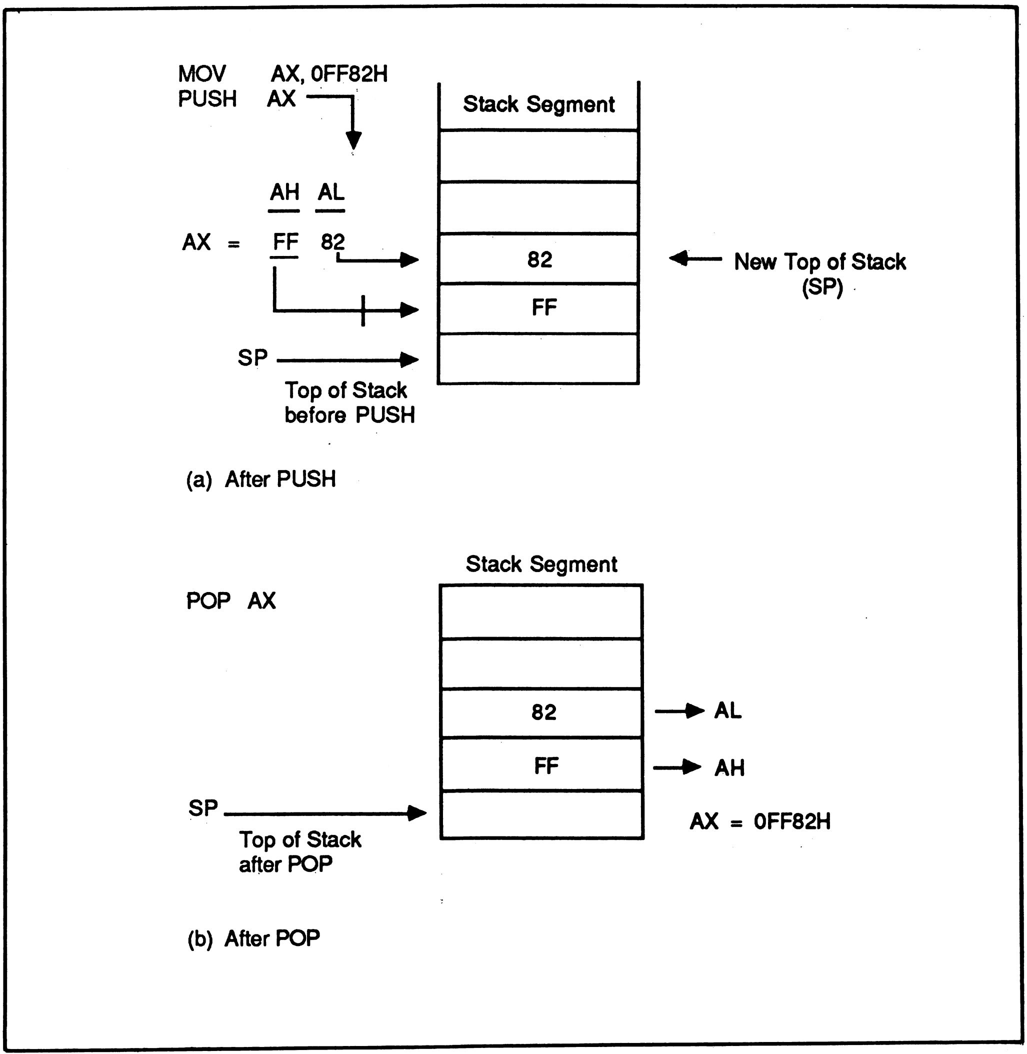

MOV AX, 0FF82H
PUSH AX
AH AL
AX = FF 82
Stack Segment
82
FF
New Top of Stack (SP)
SP
Top of Stack before PUSH
(a) After PUSH
POP AX
Stack Segment
82
FF
AL
AH
AX = 0FF82H
SP
Top of Stack after POP
(b) After POP

Figure 4-1 Stack usage during PUSH and POP

XCHG

Format: **XCHG destination,source**

The exchange instruction is used to exchange either byte or word data between the source and destination operands. Legal combinations of the XCHG instruction are:

```
XCHG        AX,CX                ;Exchange the contents of AX and CX

XCHG        AL,CL                ;Exchange the contents of AL and CL

XCHG        AL,MEMORY_8BITS      ;Exchange 8 bits (byte) of AL and

                                 ;the contents of a memory location

XCHG        AX,MEMORY_16BITS     ; Exchange 16 bits (word) between

                                 ;AX and memory.

XCHG        AX,[SI]              ;Exchange the contents of AX and

                                 ;the word in memory pointed to by SI
```

You cannot use the segment registers as the operands of the XCHG instruction. For example:

```
XCHG CS,DS                       ;Is illegal

XCHG DS,MEMORY_16BITS            ;Is also illegal.
```

XLAT

Format: **XLAT translation table**

The translate instruction replaces the contents of the AL register with a byte from a translation table. The offset into the table is assumed to be in AL, and the base address (start address) of the table must be in the BX register prior to using the XLAT instruction. This instruction is commonly used in character code translations. For example, if you want to convert an ASCII character to Baudot, you could use the XLAT instruction to perform the conversion.

I/O Port Transfers

IN and OUT

Format: **IN accumulator,port**
OUT port,accumulator

The IN instruction transfers either byte or word values from a specified port number to the accumulator. Conversely, the OUT instruction will send or transfer data from the accumulator to the specified port number. The port number can either be specified as part of the instruction (an immediate 8-bit value), or it can be specified indirectly by placing the port number in the DX register. Immediate port addressing allows up to 256 ports to be accessed (0–0FFH), while using the indirect method allows up to 65,535 ports to be accessed (0–0FFFFH).

The IBM PC communicates to various peripheral devices such as the keyboard, video display, cassette, and disk drives via port addresses. For example, port addresses 3F0–3F7H are assigned to the diskette controller. Ports 378–37AH are used to communicate with the printer interface, as are ports 3BCH–3BEH when you are using the printer port supplied on the monochrome display and printer adapter board (see the IBM technical manual for more details). Some examples of how the IN and OUT instructions may be used are:

```
PRINTER EQU 3BCH                ;Printer port on monochrome display and

                                ;printer adapter board.

PRINTER_STAT EQU 3BDH           ;Printer status port.

MOV DX,PRINTER                  ;Port address to DX

MOV BX,PRINTER_STAT             ;Printer status port into BX

MOV AL,'A'                      ;Move ASCII letter A into AL

OUT DX,AL                       ;Send character in AL to the printer.

XCHG DX,BX                      ;Exchange BX and DX. DX now =status port

IN AL,DX                        ;Get the printer's status.

Other examples IN and OUT are:

IN AX,0A0H                      ;Input a word of data from port A0H

IN AL,0B1H                      ;Input a byte of data from port B1H

OUT 0B3H,AX                     ;Output the word in AX to port B3H.
```

Address-Object Transfers

Address-object transfers send address pointers to a specified register or pair of registers. The address pointers are then used as segment and offset pointers in calculating absolute memory addresses.

LEA

Format: LEA destination,source

The load effective address (LEA) instruction is used to transfer the offset address of the source operand to the specified 16-bit register. The source register is always a memory operand. You could also use the MOV instruction with the offset operator to move an address value rather than the contents of a memory location into a register. For example:

```
     MOV SI,OFFSET SOURCE_DATA

and - LEA SI,SOURCE_DATA
```

accomplish the same task; transferring the address offset of SOURCE_DATA to the SI register.

String instructions, which will be discussed later in this chapter, require SI and DI to point to memory addresses. LEA is used to move address offsets into the registers. LEA is preferred over the MOV with offset qualifier, as it is perfectly clear when LEA is used that you are moving an address and not the contents of the source operand to a register. Additionally, LEA allows you to specify an address value indirectly, using the contents of another register. For example:

```
LEA BX,TRANS_TABLE[BP][DI]
```

could be used to load the address of an entry into a translation table. In this instance the address value is the sum of the memory address TRANS_TABLE plus the contents of BP and DI. If BP contained 3, and SI contained 4, then the effective address is equal to TRANS_TABLE + 7, or the address of the eighth entry in the table. The MOV with Offset operator does not allow this flexibility in accessing the source operand.

LDS

Format: **LDS destination,source**

The LDS (load pointer using DS) instruction transfers a 32-bit double word segment and offset value from the source operand to DS and the specified 16-bit register. For example:

```
LDS  SI,MY_DATA
```

loads SI with the offset value of MY_DATA and DS with the segment address of the segment where MY_DATA resides. This instruction is commonly used to switch the DS register to a new data segment and establish a pointer in SI prior to using string instructions which require this register combination to access data. Rather than loading AX with the start address of a new data segment and transferring the address value from AX to DS, LDS accomplishes the same function in one instruction. Additionally, it establishes an offset value that would otherwise require yet another instruction.

Like LEA, LDS allows you to add a displacement to the source operand:

```
MOV DI,05                 ;Move the value of 5 into DI

LDS BX,MY_DATA[DI]        ;Move the double word stored at

                          ;MY_DATA + 5 into DS and BX.
```

LES

Format: **LES destination,source**

LES (Load pointer using ES) operates in the same manner as the LDS instruction, with the exception that the segment address is transferred to the extra segment register, ES, rather than the data segment register, DS. Similar to LDS, which establishes source pointers to be used by the 8088's string instructions, LES can be used to set up the ES and DI registers with pointers to the destination string:

```
LES DI,YOUR_DATA                   ;Loads ES with the segment address of

                                   ;YOUR_DATA, and DI with the offset

                                   ;address of the operand.
```

Flag Register Transfers

The 8088 contains instructions that allow the programmer to transfer a portion of the 16-bit flag register to or from the high order half of the accumulator or to transfer the flag bits to or from the stack.

LAHF

Format: **LAHF**

LAHF (Load register AH from Flags) transfers the carry flag (CF), parity flag (PF), auxiliary carry flag (AF), zero flag (ZF), and sign flag (SF) into bit positions 7, 6, 4, 2, and 0 of the AH register. You may wonder why these are the only flags transferred to AH. The reason is to retain compatibility with the older 8080 and 8085 8-bit processors from Intel; these are the status flags used by the 8080/8085.

SAHF

Format: **SAHF**

SAHF (Store register AH into Flags) transfers bits 7, 6, 4, 2, and 0 from AH to the status flag register. Again, this instruction was added to the 8088 to insure compatibility with the 8080 and 8085 processors.

PUSHF and POPF

Format: **PUSHF**
POPF

PUSHF (Push Flags) saves the flag bits in the 16-bit flag register onto the stack. The stack pointer is decremented by two, and the contents of the flag register are saved at the location pointed to by SP. POPF transfers the flag bits contained in the 16-bit value pointed to by the stack pointer to the appropriate bit positions of the flag register. SP is then incremented by two to point to the new top of the stack.

You would use the PUSHF instruction whenever it is necessary to preserve the current contents of the flag register.

Consider the following code:

```
MOV  AL,BYTE_VAL                        ;Move a byte into AL

CMP  AL,BL                              ;Is it lower or higher?

CALL PROCEDURE_1                        ;Go do something with AL

JB   LOW_VAL                            ;AL BL then jump.
```

What's wrong with the program example above?

It's a pretty good bet that the procedure will use an instruction or two that will change the flags. If the flag register is not saved on entering the procedure and restored prior to exiting the routine, the flags will not be accurate when control is returned to the main program. PUSHF and POPF can be used to save and restore the flags on entering and exiting the procedure.

Program errors such as the one above are common to novice programmers. A better solution to the problem is to keep a simple rule in mind: Immediately follow the flag setting instruction with a conditional transfer instruction like this:

```
MOV  AL,BYTE_VAL             ;Fetch value from memory

CMP  AL,BL                   ;Is AL<BL?

JB   LOW_VAL                 ;Go to another routine if it is.

CALL PROCEDURE_1             ;Else go do something with AL.
```

Arithmetic Instructions

With this group of instructions, the 8088 is able to perform addition, subtraction, multiplication, and division. The operands may be signed or unsigned byte or word values. All instructions treat the data operands as binary values, although corrective instructions allow you to perform arithmetic operations on ASCII and BCD data.

The carry, auxiliary carry, zero, sign, parity, and overflow flags are set or reset according to the result of the operation. See the section on the flag register in Chapter 2 for a description of the behavior of each of these flags.

Addition

ADD

Format: **ADD destination,source**

The ADD instruction adds the source to the destination operand and places the result into the destination operand. The operands may be byte or word values; they may both be registers, or they may be a memory and register combination. The operands may be signed or unsigned binary data. Some examples of the ADD instruction are:

```
ADD     DL,3FH                 ;Add 63 to the contents of the DL register.
                               ;(DL=>DL+3FH)
ADD     AX,BX                  ;Add the contents of BX to AX(AX=>AX+BX)
ADD     WORD_VAL,DX            ;Add the contents of DX to memory location
                               ;WORD_VAL. (WORD_VAL=WORD_VAL+DX)
ADD     BL,BYTE PTR WORD_VAL   ;ADD the low order byte of the
                               ;word stored at WORD_VAL to BL.
```

In this example, the PTR operator was used to specify a byte value stored at WORD_VAL. This is a type override; it is used to override the type attribute of an operand. Each operand, when defined, has an attribute associated with its definition. In this particular case, WORD_VAL has a type attribute of word; it is a 16-bit, or 2-byte, value. In order to operate on either low order half of WORD_VAL, it is necessary to use the pointer override prefix.

If you were to code the instruction as ADD BL,WORD_VAL, the assembler would generate an error, because the attribute types of the operands do not match. I'll discuss this and other override prefixes in the next chapter. I did, however, want to show you that 1 byte of a word operand can be isolated and added to another 8-bit operand.

ADC

Format: ADC destination,source

ADC (add with carry) is used to add the source to the destination operand and add 1 to the result if the carry flag is set. The result is placed in the destination operand. The operands may be signed or unsigned binary data.

Assume for a moment that you want to add the two 32-bit numbers, as illustrated in Figure 4-2. The destination operand(s) will be in memory at WORD_ONE (low order 16 bits) and at WORD_TWO (high order 16 bits). The 32-bit operand you want to add to these memory locations is in the AX (low order word) and the BX (high order word) registers. The following set of instructions produces the correct result:

```
ADD     WORD_ONE,AX            ;Add AX to WORD_ONE without carry
ADC     WORD_TWO,BX            ;Add BX and Carry bit to WORD_TWO.
```

Add 70319028 (Hex) and 3AC5E045 (Hex)

1st Operand (High Order) - 7031H = 0111 0000 0011 0001 (Binary)
2nd Operand (High Order) - 3AC5H = 0011 1010 1100 0101 (Binary)
1st Operand (Low Order) - 9028H = 1001 0000 0010 1000 (Binary)
2nd Operand (Low Order) - E045H = 1110 0000 0100 0101 (Binary)

```
Add Low Order Operands:        1001 0000 0010 1000
ADD AX,BX                      1110 0000 0100 0101
                               -------------------
                 Carry --> 1   0111 0000 0110 1101

Add High Order Operands:       0111 0000 0011 0001
(With Carry)                   0011 1010 1100 0101
                                                 1
                               -------------------
                               1010 1010 1111 0111

ADC CX,DX

Result in CX:AX              CX                        AX
                   1010 1010 1111 0111       0111 0000 0110 1101
                        2AF7 (Hex)                706D (Hex)

                   Check Binary Add With Hex
                             Add
                        70319028 (Hex)
                      + 3AC5E045 (Hex)
                        --------
                        AAF7706D (Hex)
```

Figure 4-2 32 bit ADD,ADC Example

As the ADD instruction produced a carry out of bit 15, the ADC accounts for the carry and adds 1 to the high order word of the result. Multiple precision addition is possible using the ADC instruction.

INC

Format: INC destination

The increment instruction is used to add 1 to the destination operand. The increment instruction does not affect the carry flag (CF), but it does update AF, OF, PF, SF, and ZF. The operand may be either an unsigned byte or word value. Some examples of the INC instruction are:

```
INC     WORD_VAL                ;Increment the 16 bit contents of
                                ;WORD_VAL
INC     BYTE_VAL                ;Increment the 8 bit contents of
                                ;BYTE_VAL
```

continued

```
INC     CX                      ;Increment the value in CX

INC     AL                      ;Increment AL
```

The increment instruction adds 1 to an operand faster than the ADD instruction does. The maximum execution time is encountered when incrementing an operand in memory, yet even this is faster than ADD MEM_VAL,1. Compare the execution times for these operations as given in Appendix B.

AAA

Format: AAA

AAA (ASCII Adjust for Addition) corrects the result in AL after adding two unpacked BCD numbers. In Chapter 1, I demonstrated that after adding two BCD digits, the result may not be correct. The CPU treats all operands as binary data. We formulated the rule that if the result in the low order nibble was greater than 9, or if the auxiliary carry flag was set, you must add 6 to the low order nibble of the result to correct for the addition.

If the above conditions are met, AAA adds 6 to the low order nibble of AL, clears the high order nibble of AL, and adds 1 to the contents of AH. This in effect produces an unpacked BCD value in AH and AL. Consider the following statements:

```
MOV     AX,0009                 ;AL= 09 and AH= 00

ADD     AL,04H                  ;AL now = 0DH which is not BCD digit

AAA                             ;AL now = 03 and AH= 01(AX=0103).This is

                                ;the correct result of 9+4 = 13 (decimal).
```

DAA

Format: DAA

DAA (Decimal Adjust for Addition), corrects the value in AL from a previous addition of two packed BCD operands. If the value in AL is greater than 9 or if the auxiliary flag is set, 6 is added to AL and AF is set to 1. Also, if AL is greater than 9FH or if CF is set, then 60H is added to AL, thereby correcting the high order nibble of AL, and CF is set to 1. The following examples illustrate the DAA instruction:

```
Example 1

MOV     AL,39H                  ;AL = 39 BCD

ADD     AL,18H                  ;AL = 51H, the desired result should be 57

                                ;BCD.
```

```
DAA                             ;AL now = AL + 06 = 57


Example 2:

MOV     AL,79H                  ;AL = 79 BCD.

ADD     AL,39H                  ;Add 79 + 39. AL now = B2H

DAA                             ;Add 06 to AL, AL now = B8H

                                ;Also AAA adds 60H to B8H, AL now = 18 and

                                ;the carry flag is set to 1.
```

In Example 2, 79 BCD plus 39 BCD should yield a result of 118 BCD. Due to the binary addition of 79H and 39H, the result is B2H. DAA compensates for this error in packed BCD addition. The carry flag, which is set due to the carry out of bit 7 of AL, can be used to increment another register or memory operand to yield the three-digit BCD result of 118.

Subtraction

SUB

Format: SUB destination,source

The subtraction instruction subtracts the source operand from the destination operand and places the result in the destination operand. The operands may be either signed or unsigned byte or word values. Some examples of the SUB instruction are as follows:

```
SUB     AL,30H                  ;Subtract 30H from AL.

SUB     BX,CX                   ;Subtract the contents of CX from the

                                ;contents of BX.

SUB     AX,WORD_VAL             ;Subtract the contents of the memory

                                ;location WORD_VAL

                                from AX.

SUB     BYTE_VAL,AL             ;Subtract the contents of AL from the

                                ;memory location BYTE_VAL.
```

SBB

Format: **SBB destination,source**

Subtract with borrow (SBB) subtracts the source operand from the destination operand and then subtracts 1 from the result if the carry flag is set. Just as the ADC instruction can be used to perform multiple precision addition, SBB performs multiple precision subtraction. An example of using SBB to perform multiple precision subtraction is:

```
SUB     AX,BX                   ;First subtract the low order 16 bits

SBB     DX,CX                   ;then subtract the high order 16 bits

                                ;and the carry bit.
```

When would the carry be set from a SUB operation? Anytime the source operand is larger than the destination. In the example above, the carry flag is set if AX is less than the contents of BX. The source and destination operands can be either signed or unsigned byte or word values.

AAS

Format: **AAS**

The ASCII Adjust for Subtraction (AAS) instruction corrects the result produced by the subtraction of two unpacked BCD numbers. AAS sets the high order nibble of AL to zero and subtracts 6 from AL and 1 from AH if the value in AL is greater than 9 or if the auxiliary flag is set. AF and CF are then set, and the high order nibble of AL is cleared. If the result in AL is 9 or less, AAS clears the high order nibble of AF and updates the flags only:

```
MOV     AL,08H                  ;Use 0000 1000

SUB     AL,09H                  ;subtract 9 from 8 (should yield a -1)

AAS                             ;Adjust for BCD.
```

What has occurred at the bit level is:

```
SUB    AL,09H

       0000 1000       {8 = AL}

     - 0000 1001       {9}

       ---------       ----

       1111 1111       -1
```

Don't forget how subtraction is carried out. The two's complement is performed first on the subtrahend, and then the two's complement of the subtrahend is added to the minuend:

```
     0000 1000      {8}
 +   1111 0111      {2's complement of 9}
     ---------      ------
     1111 1111      -1
```

AAS sets the high order nibble to zero and subtracts 6 from the result, since the value contained in the low order nibble of AL is greater than 9:

```
AAS
       0000 1111      0FH
   +   1111 1010      -6
       ---------      ----
       1111 1001      {ten's complement of -1}
```

Setting the high order nibble to zero yields:

```
0000 1001       in AL,
```

AF and CF are set indicating that the value contained in AL is the ten's complement of the correct answer, −1. Your program has to account for the possibility of a BCD digit appearing in ten's complement form after adjustment. Adjustments are carried out on BCD digits without regard to sign. In other words, by testing CF and AF, you can determine whether the number in AL is true BCD or a ten's complement BCD value.

DAS

Format: **DAS**

Decimal Adjust for Subtraction corrects the accumulator (AL) after subtracting two packed BCD digits. DAS checks the lower nibble of AL for a value which exceeds 9. If AL is greater than 9 or if the previous subtraction set the auxiliary carry flag (AF), 6 is subtracted from AL and AF is set. Next, the high order nibble of AL is checked for a value greater than 9. If the most significant 4 bits of AL are found to contain a value greater than 9 or CF is set, 60H is subtracted from AL and CF is set.

Just as DAA adjusts AL after the addition of two packed BCD digits, DAS corrects AL to produce a pair of valid packed BCD digits.

DEC

Format: DEC destination

The decrement instruction subtracts 1 from the destination operand. The operand may be a byte or word, or register or memory operand. DEC updates AF, OF, PF, SF, and ZF. It does not affect CF. If you need to detect a carry condition due to the subtraction, use the SUB instruction to update the carry after execution.

Examples of DEC:

```
DEC   BX            ;Decrement word in register BX

DEC   AL            ;Decrement byte in register AL

DEC   WORD_VAL[SI]  ;Decrement word in memory

DEC   BYTE_VAL[SI]  ;Decrement byte in memory
```

NEG

Format: NEG destination

The negate instruction forms the two's complement of the destination operand by subtracting it from zero. If the destination operand is a byte equal to − 128 (80H) or a word equal to − 32,768 (8000H), NEG sets OF without altering the operand. If the operand is zero, the sign of the operand is not changed and the zero flag (ZF) is set. The other flags affected are CF and SF.

Examples of NEG:

```
NEG   AL            ;perform 2's complement on the contents of AL

NEG   AX            ;perform 2's complement on the contents of AX

NEG   WORD_VAL      ;Perform 2's complement on word in memory

NEG   BYTE_VAL      ;Perform 2's complement on the byte in memory
```

CMP

Format: CMP destination,source

The CoMPare instruction subtracts the source from the destination. Unlike the SUB instruction, which places the result in the destination operand, CMP returns no result. You might wonder about the value of an instruction that performs an arithmetic operation, yet does not return a result. Casual observation may lead you to believe the instruction is of little value. However, the status flags are updated just as if a SUB instruction were executed. This allows the testing of an operand for a given value without altering the operand.

The CMP instruction is followed by a conditional jump instruction to pass control to another part of the program, based on the results obtained from the comparison. In a moment I'll discuss the conditional jump instructions, which are used in a program to make decisions of this type.

Examples of CMP:

```
MOV    AL,80H          ;Move 128 into AL

CMP    AL,DL           ;Compare AL to DL

JC     THERE_PROG      ; If AL <DL> Jump to the portion

                       ;of the program labeled THERE_PROG

JZ     HERE_PROG       ; If AL = DL then jump to HERE_PROG

                       ;Otherwise AL is greater than DL
```

More examples of CMP:

```
CMP    BL,02H          ;Compare BL to 2 and set flags

CMP    DX,WORD_VAL     ;Compare register to memory

CMP    BYTE_VAL,0FH    ;Compare memory to 15
```

Multiplication Instructions

There are three instructions designed to carry out multiplication on the 8088. Two of these instructions accommodate either signed or unsigned multiplication; IMUL is used when signed multiplication of two operands is to take place, and MUL is used to multiply two unsigned operands. In either case, the multiplicand must be in the accumulator (AL for byte × byte multiplication, and AX for word × word multiplication).

The source operand can be either a byte or a word. If the operand is a byte operand, it is multiplied by register AL. Since byte multiplication can result in word length results, the result is placed in the AX register. AH will contain the high order byte of the result, and AL will contain the low order byte of the result.

Similarly, a word multiplication can result in a 32-bit (double word) result. In this instance, the high order word of the result is placed in DX, and the resultant low order word is placed in AX. Figure 4-3 illustrates register usage for the multiplication instructions.

Format: MUL source-operand
IMUL source-operand

8 Bit Multiplies: AL = Multiplicand
Source Operand = Multiplier
Product Returned to AX
AL = Low Order 8 bits and AH = High Order 8 bits.

For the MUL instruction, if CF and OF are set, then AH contains a non-zero result.
For the IMUL instruction, if CF and OF are set then AH is NOT the sign extension of the low order half of the result.

16 Bit Multiplies: AX = 16 bit Multiplicand
Source Operand = 16 bit Multiplier
Product Returned to DX:AX
AX = Low Order 16 bits and DX = High Order 16 bits.

For the MUL instruction, if CF and OF are set then DX contains a non-zero result.
For the IMUL instruction, if CF and OF are set then DX does NOT contain a sign extension of the low order half of the result.

Figure 4-3 Register Usage for Multiply Instructions (MUL, IMUL)

MUL

Format: **MUL source**

MUL is used to multiply two unsigned byte or word operands. MUL sets CF and OF when the high order half (AH for byte multiplication and DX for word multiplication) of the result is nonzero. In other words, the multiplication produced a result larger than 8 bits. If CF and OF are cleared, the high order half of the result is zero. You cannot use an immediate value as the source. AF, PF, SF, and ZF are undefined following a MUL operation.

Examples of MUL:

```
MUL     BX              ;Multiply AX by BX, result in DX:AX

MUL     CL              ;Multiply AL by CL, result in AH:AL

MUL     WORD_VAL        ;Multiply AX by memory word, result in DX:AX

MUL     BYTE_VAL        ;Multiply AL by memory byte, result in AH:AL
```

IMUL

Format: IMUL source

The Integer MULtiply instruction is used to multiply two signed byte or word operands. The destination operand is the accumulator, AL or AX. Like the MUL instruction, the result is returned to either AH and AL for byte multiplication or DX and AX if the operands are word values.

CF and OF are set following the IMUL instruction if the most significant half of the result is not a sign extension of the lower half of the result. This indicates that the multiplication produced a result in excess of 8 bits for byte-wide multiplication or in excess of 16 bits for word-wide multiplication, and that AX or DX contain the high order half of the result.

What Are Sign Extensions?

When multiplication is carried out, byte-wide multiplications can produce a signed 16-bit (word) result. Similarly, a word-wide multiplication can produce a signed 32-bit, or double word, result. The sign of the result is extended to the high order half of the result (AH or DX) when the product can be contained within the low order half of the result register(s). In other words, when the multiplication produces a product in the range of −128 to +127, or −32,768 to +32,767, the sign bit is extended to the high order half of the result.

For example:

```
High Order Byte (AH)          Low Order Byte (AL)

  1111 1111          <<--     1000 0000   (-127)

  1111 1111          <<--     1111 1111   (-1)

  0000 0000          <<--     0000 0001   (+1)

  0000 0000          <<--     0111 1111   (+128)

High Order Word (DX)          Low Order Word (AX)

1111 1111 1111 1111  <<--     1000 0000 0000 0000     (-32,768)

1111 1111 1111 1111  <<--     1111 1111 1111 1111     (-1)

0000 0000 0000 0000  <<--     0000 0000 0000 0001     (+1)

0000 0000 0000 0000  <<--     0111 1111 1111 1111     (+32,767)
```

In each instance, the sign bit of the low order half is copied into each bit of the high order byte, thereby preserving the true value of the result.

By extending the sign of an operand, different data types may be used in signed multiplication, addition, and division operations. The 8088 instructions CBW (convert byte to word) and CWD (convert word to double word) perform sign extensions on the accumulator. CBW copies the sign bit (b7) of AL into each bit position of AH. CWD copies the sign bit (b15) of AX into each bit position of the DX register.

I'll discuss how the CWD is used in division in a moment, but for now consider the following examples:

Example 1.
Add the byte value of 16H to the word value in CX:

```
MOV     AL,16H          ;Place the value into AL

                        ;AL= 0001 0110 and AH = ???? ????

CBW                     ;AH = 0000 0000

                        ;AL = 0001 0110

ADD     CX,AX           ;Add AX to CX
```

Example 2.
Multiply the byte value in AL by the word stored in memory at the location WORD_VAL:

```
MOV     AL,80H          ;Move -128 into AL

CBW                     ;AH = 1111 1111 and AL = 1000 0000

IMUL    WORD_VAL        ;Multiply by memory operand, result is

                        ;in DX:AX
```

Example 3.
Subtract a word in memory from the byte value in register BL:

```
XCHG    AL,BL           ;Get value into AL in order to perform

                        ;sign extension.

CBW                     ;Extend sign

XCHG    AX,BX           ;swap registers, restore BL, and place

                        ;extension in BH.

SUB     BX,WORD_VAL     ;A word has now been subtracted from

                        ;what was a byte value.
```

AAM

Format: **AAM**

The instruction AAM (ASCII Adjust for Multiplication), corrects the result of a previous multiplication of two valid unpacked BCD digits. In order for AAM to work correctly, the high order nibbles of the operands multiplied must have been set to zero prior to executing the multiplication instruction. AAM returns two valid unpacked BCD digits to AH and AL.

AAM performs the correction by first dividing AL by 10. The quotient is placed in the AH register, and the remainder is placed into the AL register. AAM affects PF, SF, and ZF. AF, CF, and OF are undefined following the execution of the AAM instruction.

Division

Three instructions are used for division.

DIV

Format: **DIV source**

The DIV instruction is used when performing an unsigned division of the accumulator by the source operand. If the source operand is a byte value, it is divided into the 16-bit value (word) contained in AH and AL. The quotient is returned to AL, and the remainder is placed in AH.

Similarly, if the source operand is a word value, it is divided into the 32-bit (double word) value contained in DX and AX. The quotient is placed in AX, and the remainder is returned to DX. The source operand cannot be an immediate value (constant).

Division by Zero

As I discussed previously one of the interrupt vectors (type zero) is used for an error condition, division by zero. Since division by zero is undefined, the 8088 generates a type zero interrupt. To avoid this condition, you should test the source operand prior to executing the DIV or IDIV instructions. You could also replace the vector used for the type 0 interrupt with one containing the address of your own error routine.

The type zero interrupt is executed anytime the capacity of the destination register is exceeded by the quotient. For unsigned divisions using the DIV instruction, this condition exists when the quotient exceeds 0FFH (256) for byte source operands and 0FFFFH (65,535) for word operands.

When these values are applied to signed operands using the IDIV instruction, they become quotients where maximum and minimum values are exceeded. A type zero interrupt is executed when byte operands exceed 7FH (+127) and 80H (−128). A type zero interrupt is also executed when word size source operands exceed 7FFFH (+32,767) and 8000H (−32,768).

When the type zero interrupt occurs, the quotient and remainder are left in an undefined state. The DIV instruction leaves AF, CF, OF, PF, SF, and ZF in an undefined state after execution.

Some examples of DIV are:

```
DIV     BL              ;Divide AX by BL, quotient to AL, remainder to AH

DIV     CX              ;Divide DX:AX by CX, quotient to AX, remainder to

                        ;DX

DIV     BYTE_VAL        ;Divide by byte wide memory operand.

DIV     WORD_VAL        ;Divide DX:AX by the contents of the word wide

                        ;memory operand WORD_VAL.

MOV     CL,10H          ;Divide by a constant (16 decimal).

DIV     CL              ;Divide AH:AL by the contents of CL (=16)
```

IDIV

Format: IDIV source

The Integer DIVide instruction performs division on the signed source operand and a signed value in either AH and AL (source = byte) or DX and AX when the source operand is a byte value.

As in the DIV instruction, a type zero interrupt is executed anytime the maximum or minimum capacity of the destination operand is exceeded by the quotient (see the above discussion of division by zero). This can occur when the source operand is zero.

Some examples of IDIV are:

```
IDIV    CL              ;Divide AH:AL by CL. Quotient in AL and remainder

                        ;in AH

IDIV    CX              ;Divide DX:AX by CX. Quotient in AX and remainder
```

```
                             ;in DX.
IDIV    BYTE_VAL[SI]         ;Divide AH:AL by byte value at memory
                             ;BYTE_VAL + the offset in SI.
MOV     BL,10H               ;divide AH:AL by 16
IDIV    BL
```

I mentioned that the CWD (Convert Word to Double word) instruction can be used to sign extend (copy) a word value to a double word value. The sign bit, bit 15, of the word in AX is sign extended into all bit positions in the DX register. By using CWD and IDIV, you can divide two-word values.

IDIV expects to find a 32-bit dividend in DX and AX. You cannot simply move a signed value into the AX register and then divide by some other value. DX must contain either the high order 16 bits of the dividend or a sign extension of the 16-bit dividend contained in AX.

Some examples of sign extension division are:

```
MOV     AX,WORD_VAL1         ;Move 16 bit value into AX
CWD                          ;Sign extend AX to DX
IDIV    WORD_VAL2            ;Divide by the contents of memory
                             ;location WORD_VAL2
MOV     AL,BYTE_VAL1         ;Get dividend
CBW                          ;Sign extend to AH
IDIV    BYTE_VAL2            ;Divide by the contents of memory
```

AAD

Format: **AAD**

The ASCII Adjust for Division corrects the dividend in AL prior to dividing two unpacked BCD numbers. Essentially, the instruction multiplies the AH register by 10 and adds this value to the low order register AL. In effect, the packed value is converted to binary. AH is then zeroed:

```
MOV     AX,0309              ;In unpacked BCD this value represents
                             ;39 decimal.
AAD                          ;Value in AX is now- AH = 0000 0000
                             ; and AL = 0010 0111
IDIV    CL                   ;divide by the contents of CL.
```

AAD updates PF, SF, and ZF. All other flags are undefined after executing the AAD instruction.

Before You Continue

If you decided to read this chapter straight through, I suggest that you now take a break, go get a cup of coffee (etc.), and, by the time you get back, I'll be ready to talk about logic instructions. We have much more to discuss in this chapter, so take a break; you deserve one.

Bit Manipulation Instructions

Logic Instructions

The logic instructions apply Boolean logic to each bit position of the source operand and to the corresponding bit positions of the destination operand. Whether you know it or not, the rules of logic abound in everyday life. Let's take a look at some everyday events and relate this logic to the logical instructions AND, OR, XOR, TEST, and NOT.

AND

***Format:* AND destination,source**

A logical AND function is one which states that two events must be true in order for the result of the operation to be true. Table 4-2 contains the truth table for the AND function.

If I want to turn my IBM PC on, I must have the power cord plugged into an active AC outlet AND I must turn the power switch on. If either of the input conditions is not satisfied, power is not delivered to the internal power supply of the IBM PC. An

Table 4-2
Truth Table For Logic Operations

AND & TEST

Operands A	Operands B	Result C
0	0	0
0	1	0
1	0	0
1	1	1

Example: AND AL, BL ;AL=07EH
;BL=023H

AL = 0111 1110
BL = 0010 0011

Result in AL = 0010 0010 (After AND only)
*TEST does not return the result to the
*destination operand.

OR

Operands A	Operands B	Result C
0	0	0
0	1	1
1	0	1
1	1	1

Example: OR BL, CL ;BL=34H
;CL=E7H

CL = 1110 0111
BL = 0011 0100

Result in BL = 1111 0111 = 0F7H

XOR

Operands A	Operands B	Result C
0	0	0
0	1	1
1	0	1
0	0	0

Examples: XOR AH,34H ;Find bits different

AH = 1101 0011
0011 0100

Result in AH = 1110 0111 = E7H = AH

Clear AX: XOR AX,AX
AX = 1010 0011 1111 0101
1010 0011 1111 0101

Result in AX -> 0000 0000 0000 0000

NOT

Operand A	Result B
1	0
0	1

Example: NOT AX
AX = 1101 0011 1101 0111

NOT --> 0010 1100 0010 1000
(Result in AX)

equation of this function can be written as:

(Active AC power outlet) AND (Power switch in the ON position) = IBM PC in the ON condition.

When using the AND instruction in a source program, you are performing a logical AND of either a byte or word source operand with that of the destination. Each bit in the source is ANDed with the corresponding bit of the destination. Bit 1 is ANDed with Bit 1, b2 with b2, etc. The result of the operation is placed in the destination operand.

For example, AND AL,72H functions in the following manner:

```
                         b7 b6 b5 b4 b3 b2 b1 b0

AL= 0F0H AND 72H = 1  1  1  1  0  0  0  0 (Binary)

             AND 0  1  1  1  0  0  1  0 (Binary)

                   -------------------------

 Result is in AL = 0  1  1  1  0  0  0  0 (Binary) = 70H
```

Notice that bit7 is reset in the result. When b7 of the destination (AL), which contains a 1, is ANDed with the zero contained in b7 of the source, b7 of the result is zero due to the rules of logic illustrated in Table 4-2. Similarly, b1 is also reset in the result.

Some examples of the AND instruction are:

```
AND     AL,BL           ;AND the contents of AL with BL and

                        ;place the result in AL.

AND     BX,DX           ;AND two 16 bit registers.

AND     WORD_VAL,AX     ;AND a register and memory, place result

                        ;in memory.

AND     AL,BYTE_VAL     ;AND the contents of memory with AL and

                        ;place the result in AL.

AND     AL,11111000B    ;AND AL with a constant
```

AND updates CF, OF, PF, SF, and ZF. AF is undefined following the AND instruction.

You can use the AND instruction to mask bit position. A mask is a predefined bit pattern used to set or reset certain bit positions. In the example above, AL was ANDed with the binary pattern 11111000. This has the effect of clearing bits 0–2 in the destination operand (AL), while leaving the other bit positions in whatever state they were in prior to the execution of the AND instruction.

OR

Format: **OR destination,source**

The OR instruction sets a bit in the result, provided that the corresponding bit in either the destination OR the source operand is set to a binary 1. It is a little like saying I'll wake up in the morning if my alarm clock goes off OR if someone wakes me; otherwise, I'll sleep until noon. If either situation is satisfied, independently or simultaneously, I will awaken. Table 4-2 contains the truth table for a two-input OR function.

Some examples of the OR function are:

```
OR      AL,11110000B    ;OR AL with a constant. This technique
                        ;can be used to set specific bits in the
                        ;destination operand while leaving all
                        ;others in their previous state. Here
                        ;bits 4-7 are set to a '1'
OR      BYTE_VAL,AL     ;OR memory with contents of AL
OR      AX,BX           ;OR AX with the contents of BX.
OR      BX,WORD_VAL     ;OR BX with word in memory.
```

The OR instruction updates CF, OF, PF, SF, and ZF. AF is undefined after execution of the OR instruction.

The primary use of the OR instruction is to set specified bits in the destination operand. However, the OR instruction can be used to determine if a register contains zero. For example, OR AX,AX sets the status flags but does not alter any bits of the AX register, since a bit ORed with itself remains unchanged (A OR A = A). If AX is zero, the Z flag is set after the instruction OR AX,AX. You can use this information to effect a conditional jump (JZ, jump on zero) to another part of the program.

XOR

Format: **XOR destination, source**

The eXclusive OR instruction sets a bit in the result if the corresponding bit in the source and destination operands are of opposite values (one set to 1, and one reset to 0). In other words, if one and only one bit for a given bit position in either the source or destination operand is true (1), then the corresponding bit position of the result will also be true. This is known as mutual exclusion. The truth table for the XOR function is given in Table 4-2.

The XOR instruction can be used to determine which bits are not the same in a source and destination operand. Bits that differ between the source and destination operand are set in the result following the XOR. The XOR instruction can also be used to clear a register to zero.

For example, XOR AX,AX zeros the AX register. Since the bits in both the destination and the source are the same (it's the same register), the resultant bits are all zero. XOR updates CF, OF, PF, SF, and ZF. AF is undefined after the execution of the XOR instruction.

Some examples of the XOR instruction are:

```
XOR     AL,AL           ;Set AL to zero

XOR     AL,11000000B    ;If bits 7 or 6 are set in AL, flip them to

                        ;zero.

XOR     BX,AX           ;Set the bits which are different in original
                        ;operands.
```

TEST

Format: TEST destination,source

The TEST instruction performs a logical AND on the destination and source operands. Unlike the AND instruction, which returns the result of the operation to the destination operand, the TEST instruction does not return a result. It does, however, update CF, OF, PF, SF, and ZF. AF is undefined after the TEST instruction has been executed.

Some examples of the TEST instruction are:

```
TEST    AL,00001111B    ;Test the AL register, bits 0-3 for

                        ;a logical 1.

JNZ     THERE           ;If any bits are set jump to

                        ;another part of the program.

TEST    DX,BX           ;AND the contents of the word wide

                        ;registers DX and BX and set flags.

TEST    PORT_DATA,AX    ;AND memory and AX, and set flags
```

NOT

Format: **NOT destination**

The NOT instruction performs the one's complement on the destination operand. Each bit in the destination operand is reversed. No flags are affected by the NOT instruction.

Some examples of the NOT instruction are:

```
NOT     AL              ;Flip bits in AL

NOT     WORD_VAL        ;Perform 1's complement on the word stored in

                        ;memory.

NOT     BYTE_VAL        ;Perform 1's complement on memory operand.

NOT     DX              ;Invert the bits in DX.

NOT     ME ;Complement memory. (I couldn't resit this

;one!).
```

Shift Instructions

The shift and rotate instructions are used to isolate individual bits within a byte or a word. Each affects the carry flag as shown in Figure 4-4. These instructions can also be used to multiply or divide an operand by a power of two. For example, if the value contained in AL is 8, shifting AL left one bit position has the effect of multiplying AL by 2. This can be illustrated as:

```
AL=8 decimal = 0000 1000

Multiply by 2:  SAL AL,1 then AL = 0001 0000 = 16 decimal.

Divide by 2 :   SAR AL,1 then AL = 0000 1000 = 8 decimal.
```

Shift instructions shift a bit out of the destination operand and into the carry. Unlike the shift instructions where the MSB or LSB is shifted into the carry, the rotate instructions recirculate the bit back into the destination operand. In this manner, the bit is preserved.

SAR

Format: **SAR destination, count**

As shown in Figure 4-4, Shift Arithmetic Right (SAR) and Shift Arithmetic Left (SAL)

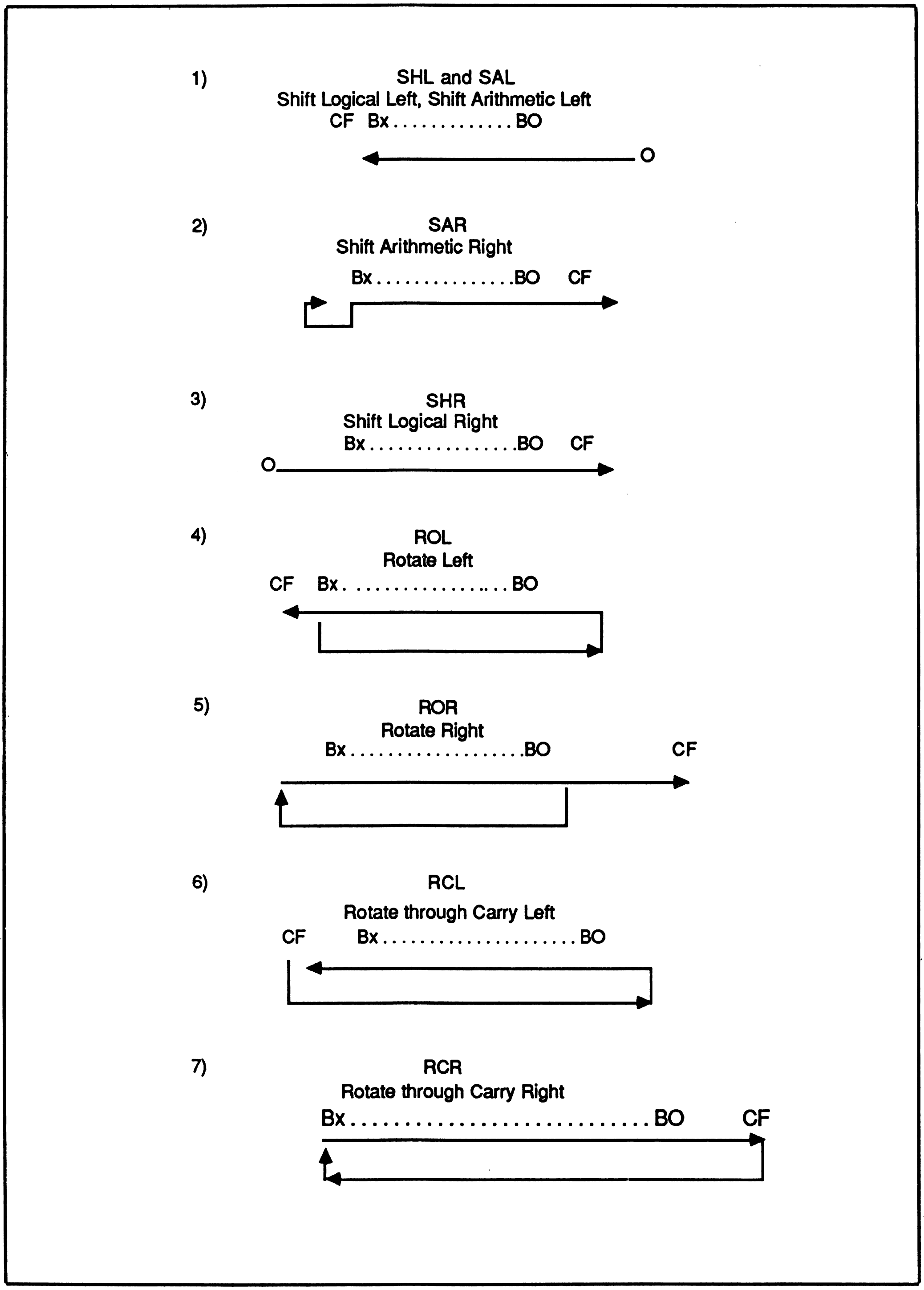

1)
SHL and SAL
Shift Logical Left, Shift Arithmetic Left
CF Bx............. BO
O
2)
SAR
Shift Arithmetic Right
Bx...............BO CF
3)
SHR
Shift Logical Right
Bx................BO CF
O
4)
ROL
Rotate Left
CF Bx. .... BO
5)
ROR
Rotate Right
Bx...................BO CF
6)
RCL
Rotate through Carry Left
CF Bx..................... BO
7)
RCR
Rotate through Carry Right
Bx............................ BO CF

Figure 4-4 Illustration of how each bit position is shifted, or rotated the specified number of times. In the illustrations, Bx is the MSB of the operand. For byte values Bx would be B7 and for word values, Bx would be B15. Note how the carry is used in each instruction.

are used to shift the contents of an operand right or left 1 or more bit positions. The arithmetic shifts move the operand by a specified number of bit positions. If you want to shift only one bit position to the right, the count can be specified by SAR AL,1. However if the count is greater than one, register CL must be used to specify the total number of bits to be shifted as in:

```
MOV     CL,3    ;Shift 3 bits

SAR     BL,CL   ;Shift BL three times
```

SAR shifts the low order bit into the carry, and the high order sign bit remains unchanged (b7 for byte operands and b15 for word operands). The original sign bit is shifted in from the left, which causes the sign bit to remain unchanged.

Be careful when you use the SAR instruction to divide a negative number by a power of two. SAR may not produce the anticipated result. See Appendix B and the SAR description to find out why.

SAL and SHL

Format: **SAL destination,count**
SHL destination,count

Shift Arithmetic Left (SAL) and Shift Logical Left (SHL) shift the destination operand left by a specified number of bits, as shown in Figure 4-4. These instructions are identical in operation. Both shift zero bits into the destination operand from the right, and the bit shifted out of the MSB of the operand is transferred to the carry. The overflow flag is set if a bit shift count of one produces a result whose sign differs from the original operand.

For example, if AL contains 1000 0001B, then:

```
SHL     AL,1
```

produces 0000 0010B in AL. The binary 1 shifted out of the operand (b7) is contained in the carry. Since b7, the sign bit, did not retain its original value, OF is set. The overflow flag is not defined for bit shifts of a count other than 1. PF, SF, and ZF are also affected by SHL and SAL, while AF is undefined.

SHR

Format: **SHR destination,count**

The Logical Shift Right instruction (see Figure 4-4) is used to shift the bits in the destination operand right a specified number of bit positions. As in the case of the other shift instructions, a count of 1 can be coded directly into the instruction. For variable shift counts, the contents of the CL register define the shift count. Zeros are shifted into the MSB of the destination operand, while the LSB (b0) is shifted into the carry flag.

Some examples of the SHR instruction are:

```
SHR     AL,1                    ;Shift AL right 1 bit.

SHR     BX,CL                   ;Shift BX right by the number of bits

                                ;specified in CL.

SHR     WORD_VAL[SI][BX],CL     ;Shift the contents of the memory

                                ;located at WORD_VAL plus the

                                ;offset of SI and BX, right by the

                                ;number of bits specified by CL.
```

Rotate Instructions

The rotate instructions ROL, ROR, RCL, and RCR are used to rotate bits in a circular fashion from left to right or right to left. Figure 4-4 illustrates the manner in which the bits in an operand are rotated for each instruction.

Unlike the shift instructions, which shift bits into and out of the operand, rotate instructions actually circulate a bit from one side of the operand to the other. RCL and RCR involve the carry in the rotation of bits, whereas ROR and ROL copy the MSB or LSB into the carry. Assume that BL contains 0001 0011B and the carry bit is set. After executing the instruction:

```
RCL BL,1
```

BL contains 0010 0111B, and the carry flag contains zero, the value shifted out of b7 from the operand.

ROR, ROL

Format: **ROR destination,count**
ROL destination,count

ROtate Right (ROR) is used to shift the destination operand a specified number of bit positions to the right. ROL is used to shift the contents of the operand to the left by a specified number of bit positions. ROR shifts the LSB (b0) of the operand into the MSB (b7 for byte operands and b15 for word operands). The LSB is also copied into the carry flag.

Similarly, ROL shifts the MSB of the operand into the LSB of the operand, while placing a copy of the MSB in the carry flag. OF is set if the count specified is equal to 1 and the sign bit (MSB) of the operand changes value.

RCL and RCR

> *Format:* **RCL destination,count**
> **RCR destination,count**

Rotate through Carry Right (RCR) rotates the destination operand right a specified number of bit positions. (See Figure 4-4.) CF is rotated into the MSB of the operand, while the LSB becomes the new value of the carry flag.

Rotate through Carry Left (RCL) rotates the destination operand to the left a specified number of bit positions. The carry flag is rotated into the LSB, while the previous contents of the carry are rotated into the MSB of the operand.

Assume AL contains 0111 1010B = 7AH, the carry flag is initially reset, and CL contains 3. The following illustrates the RCR instruction:

```
                     C  b7 b6 b5 b4 b3 b2 b1 b0

Original Value in AL 0  0  1  1  1  1  0  1  0

    RCR    AL,CL   ; 0  0  0  1  1  1  1  0  1   (1st rotate)

                   ; 1  0  0  0  1  1  1  1  0   (2nd rotate)

                   ; 0  1  0  0  0  1  1  1  1   (3rd rotate)
```

The instruction rotates the AL register to the right 3 bit positions. The final value in AL is 8FH and the carry flag is reset.

Other examples of the rotate instructions are:

```
ROR  BX,CL              ;Rotate BX the number times specified in CL

RCL  AL,1               ;Rotate AL left one time. Include the carry ;bit.

ROL WORD_VAL[SI],CL     ;Rotate the contents of memory, CL number of times.
```

Control and Transfer Instructions

Control and transfer instructions allow branching to another part of the program; program execution control is transferred to a new portion of memory. Intel divides the control transfer group into four distinct categories:

1. Unconditional transfers;
2. Conditional transfers;
3. Iteration control; and
4. Interrupts.

Control may be passed to a portion of memory in the current segment, or control may be passed to a new code segment.

Unconditional Transfers

Unconditional transfers do not require a condition to meet prior to transferring control to another portion of the program. It's a little like the drill sergeant in the army who screams "JUMP!" and you ask "How high?" The only difference is that the 8088 needs to know "How Far?"

CALL and RET

Format: **CALL procedure**
RET optional POP value

The CALL instruction enables a program to invoke a procedure (also known as a subroutine). A procedure is a frequently used portion of code. When the instructions required to execute a certain function are incorporated into a procedure and not placed into the main program every time the procedure is needed, there is an overall reduction in the amount of code generated.

The procedure is called as required. Calls can be made to procedures within the same code segment (intrasegment) or to procedures in a different code segment (intersegment). When the 8088 encounters an intrasegment CALL instruction, the contents of the instruction pointer are pushed onto the stack, and the instruction pointer is loaded with the offset address of the procedure. Program execution then continues with the first instruction contained in the procedure.

The RET (RETurn) instruction is used to exit the procedure and return to the calling program at the instruction immediately following the CALL instruction. When the 8088 encounters the intrasegment RET instruction, the 16-bit offset which was saved on the stack is popped into the instruction pointer. This causes program execution to resume at the instruction following the original CALL instruction.

An intersegment CALL invokes a procedure contained in a code segment other than the one currently in use. An intersegment CALL first pushes the current contents of the code segment register onto the stack, followed by the contents of the instruction pointer. The segment address of the procedure is then placed into the CS register, and the procedure's offset within the segment is placed into the IP register.

When an intersegment RET is encountered within the procedure, the offset value previously pushed onto the stack is placed in the instruction pointer, and the CS register is restored with its previous value, which was also saved on the stack. Program execution resumes with the instruction following the original CALL instruction.

Figure 4-5 illustrates how an intrasegment procedure is written and how the procedure is called from another portion of the program. Notice that the procedure is defined as NEAR. This assembler pseudo-op instructs the assembler to generate intrasegment CALL and RET instructions. If the procedure had been defined as FAR, intersegment CALLs and RETs would have been generated. When an intrasegment CALL is executed, only the instruction pointer is saved on the stack. When an intrasegment RET is executed, the IP register is restored, and program execution resumes with the instruction following the CALL to the procedure.

```
MAIN_PROC PROC FAR
        ..                      ; Main program code
        ..
        ..
        ..
        ..
        CALL WHATS_MY_LINE      ; Call the procedure: Defined in
                                ; this segment. Only IP is saved!
        ..
        ..
MAIN_PROC ENDP

WHATS_MY_LINE PROC NEAR
        ..                      ; Procedure Code
        ..
        ..
        RET                     ; IP is restored by this instruction.
WHATS_MY_LINE ENDP
```

Figure 4-5 Intrasegment Procedure Example

Remember the difference: Intrasegment CALLs and RETs save and restore only the offset IP, while intersegment CALLs and RETs save and restore both the CS register and the IP register. You can directly specify the procedure's address within the instruction as in CALL THAT_PROCEDURE. THAT_PROCEDURE may be of either a NEAR or FAR type.

Additionally, you may want to place a procedure's address in memory and indirectly call the procedure by using one of the general purpose registers. To do this, the register is loaded with the address of the memory location containing the address of the procedure to be called.

For example, assume the memory location KEY_BRD holds the address for a keyboard scan routine. The routine scans the keyboard and determines if there is a key closure. The following instructions can be used to indirectly call the procedure through the BX register:

```
LEA     BX,KEY_BRD              ;Address of KEY#BRD memory location.

CALL    BX                      ;Use the address stored at KEY_BRD as
                                ;the procedure's start address.
```

It is often necessary to pass a value (or two, or more) to a procedure. The parameters may be required to perform some type of calculation or to access data (i.e., they may be an address parameter). In either case, the parameters can be pushed onto the stack prior to calling the procedure. Once in the procedure, the BP register is used with the stack pointer (SP) to address the variables.

When the procedure is finished, an optional constant can be specified in the RET instruction. The constant is added to stack pointer. Remember our rule about dealing with the stack: For every PUSH there is a POP, and for every POP there must be a PUSH. The parameters that were passed to the subroutine must be discarded. The optional constant operand effectively pops the parameters from the stack, thereby restoring the stack pointer.

JMP

***Format:* JMP target**

The JuMP instruction transfers control to another portion of code. The transfer location may be in the same segment (intrasegment) or in another segment (intersegment). JMP does not save information on the stack, as the CALL instruction does. Like the CALL instruction, the JMP instruction can be either a JMP direct or a JMP indirect. You can also use the assembler pseudo-op SHORT to specify that the target operand is within +127 to −128 bytes of the JMP instruction. The assembler automatically generates a 2-byte SHORT JMP if the target is within the allowable range. If the assembler cannot make this determination or if you do not explicitly use the SHORT operator, the assembler generates a 3-byte NEAR JMP instruction. NEAR JMPs allow the target to be within +32,767 to −32,768 bytes from the JMP instruction.

Some examples of the JMP instruction are:

```
JMP  CX                    ;Jump indirect using CX

JMP  THIS_SEGMENT          ;Jump NEAR

JMP  SHORT THIS_SEGMENT    ;THIS_SEGMENT is within +127 or -128
                           ;bytes of this instruction.

JMP FAR PTR THAT_SEGMENT   ;Jump to another segment.
```

Intersegment jumps produce five bytes of code. One byte signifies that the JMP is an intersegment JMP, two bytes specify the code segment, and two bytes specify the segment offset of the target operand. JMP does not affect the flags.

Conditional Transfer Instructions

Certain instructions set or clear the status register's flag bits. Conditional transfer instructions transfer control to another part of the program if the specified condition is met. It's a little like saying I will meet you for lunch IF you are free today at noon AND you are buying. If the conditions are met, I will be somewhere I would not otherwise be—having lunch with you.

Conditional transfer instructions behave in a similar manner. If a condition is met, the 8088 begins execution at a new address. If the condition is not met, the 8088 executes the next instruction, the one following the conditional jump instruction. This decision-making capability is why the microprocessor is so important in real-time applications. Since the microprocessor is capable of executing hundreds of thousands of instructions per second, decisions are made in microseconds.

The 8088 allows many different types of conditional transfer instructions (see Table 4-3). Included in this instruction group are instructions which will jump to another part of program memory if CF, ZF, SF, OF, or PF is set. JZ target transfers control to another part of the program (specified by the operand target), if the zero flag is set, while JC target transfers control to another part of the program if the carry flag is set.

Also included in the conditional jump instruction set are instructions which transfer control to another part of memory when the status flags are reset. JNC target causes program execution to begin at the specified portion of memory if the carry flag is clear (CF = 0). JNZ target transfers control to the target operand if ZF = 0.

Table 4-3
Conditional Transfer Instructions

	Jump if. . . .	**Flag Condition**
JA/JNBE	Jump if above/not below	(CS or ZF) = 0
JAE/JNB	Jump if above or equal/not below	CF = 0
JB/JNAE	Jump if below/not above or equal	CF = 1
JBE/JNA	Jump if below or equal/not above	(CF or ZF) = 1
JC	Jump if carry	CF = 1
JE/JZ	Jump if equal/zero	ZF = 1
JG/JNLE	Jump if (< > / not > nor =	((SF xor OF) or ZF) = 0
JGE/JNL	Jump if greater or = /not <	(SF xor OF) = 0
JL/JNGE	Jump if less/not greater nor equal	(SF xor OF) = 1
JLE/JNG	Jump if < or = / not >	((SF xor OF) or ZF) = 1
JNC	Jump if not carry	CF = 0
JNE/JNZ	Jump if not equal/not zero	ZF = 0
JNO	Jump if not overflow	OF = 0
JNP/JPO	Jump if not parity/parity odd	PF = 0
JNS	Jump if not sign	SF = 0
JO	Jump if overflow	OF = 1
JP/JPE	Jump if parity/parity even	PF = 1
JS	Jump if sign	SF = 1

I mentioned in a previous example that the carry flag is set when a CMP instruction is used and the destination operand contains a value less than the source operand. We can state this condition in several ways. We can jump if the carry flag is set (JC), if the destination is less than the source (JB), or if the destination is not above (greater than) or equal to the source operand (JNAE). All three instructions accomplish the same function: jump to another portion of code if the carry flag is set. Similarly, the instructions JNC (Jump on No Carry), JNB (Jump on Not Below) and JAE (Jump on not below) all transfer control to another portion of the program if CF = 0.

The status flags are set or reset as a result of a previous instruction. Some of the 8088 instructions alter the flags, while others do not. The MOV instruction does not affect any of the flags; it simply moves data from one place to another within the system. CMP, SUB, ADD, SBB, ADC, and the logical, shift, and rotate instructions are among those which do alter the flags.

A Programming Example

Assume for a moment that you want to go to the part of the program which turns your coffee warmer off once the coffee has reached a preset temperature. The program flow chart might look something like that of Figure 4-6. In Chapter 2 in Figure 2- 5, the diamond-shaped symbol designates a portion of the program where

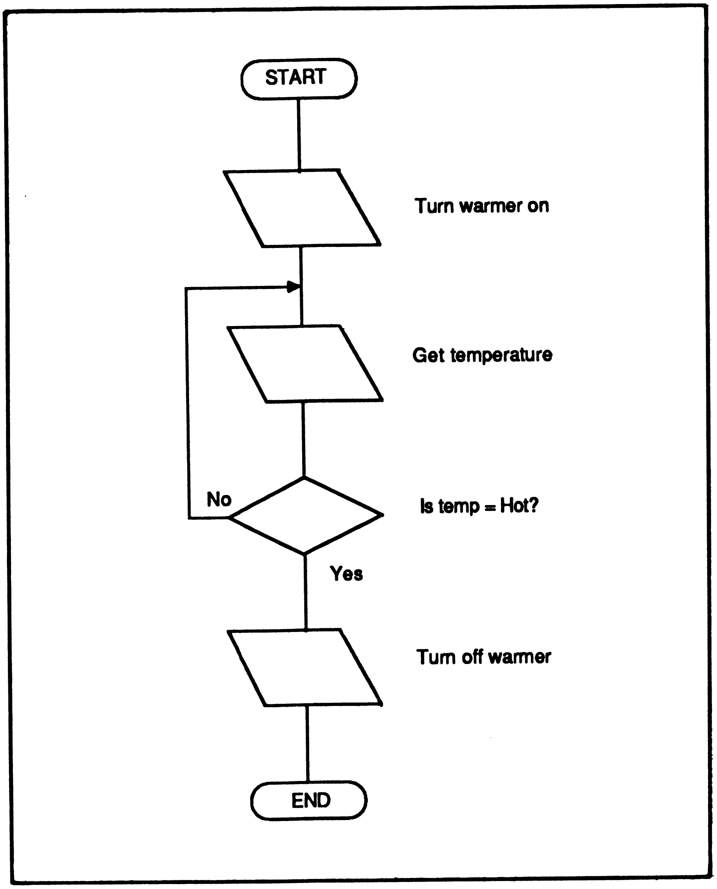


Figure 4-6

a decision must be made. This is where you would insert one or more of the conditional jump instructions from Table 4-3.

In our coffee warmer program, we'll make some assumptions about the hardware. An analog-to-digital converter (ADC) is attached to the computer at some port labeled TEMP_PORT. Another port, CONTROL_PORT, is used to turn the warmer on and off. The exact port numbers are not important in our example.

The ADC is used to convert an analog voltage to a digital byte or word that the microprocessor understands. The voltage fed to the ADC is a proportional representation of the coffee's temperature. The binary representation obtained by the ADC tracks the voltage and, therefore, the temperature.

Now all you have to do is to choose the appropriate instructions to turn the warmer on, check the temperature, and turn the coffee warmer off if the desired temperature has been reached. Assume that the binary value 0000 1111B is used to turn the warmer on and 1111 0000B turns the warmer off, and that the label HOT defines the binary representation of the temperature the coffee must be at when the warmer is turned off.

```
TEMPERATURE       PROCEDURE    NEAR

      OUT CONTROL_PORT,00001111B       ;Turn warmer on.

GET_TEMP:  IN AL,TEMP_PORT             ;Get the current temperature

      CMP AL,HOT                       ;Is the value HOT?

      JAE TURN_OFF                     ;Jump if the temperature is
                                       ;above or equal to the value
                                       ;of HOT.

      JMP GET_TEMP                     ;If it is not HOT, then test again.

TURN_OFF:  OUT CONTROL_PORT,11110000B  ;Turn the warmer off.

      RET                              ;Return to the main program

TEMPERATURE ENDP
```

This example is typical of how decision-making instructions are used within a program. Actually, the procedure above is not a very good program, because there is no provision for what to do if the desired temperature is never reached. In other words, my program is only looking for the temperature to reach a predetermined level; it does not care how long it takes.

A better approach would be to check a timer and count the elapsed time that the warmer has been on. If the temperature is not reached in a specified amount of time, the program should terminate and shut off the timer.

Using RAM for Communications

Another often used technique is to define each bit position in a byte or a word of memory, giving each a special meaning. The significance of each bit position is used to convey information to another part of the program. RAM locations are often used for such purposes and are commonly referred to as semaphores.

If the bit is set, it means one thing; if the bit is reset, is means another. The sample program at the end of Chapter 2 fetched a word from memory that contained the system configuration for your computer. Each bit in the word specifies whether a resource (such as an RS-232 card or the game port adapter) is in the system.

By rotating a bit into the carry, a decision can be made as to whether or not the resource is available. If the bit is set, the resource is available. If the bit is reset, the resource is not in the system.

For example, find the label DC2 in Listing 2-1. The status word is moved into the AX register, and b0 is rotated into the carry. If this bit is reset, there are no disk drives in the system. If the bit is set, the program determines the number of available disk drives and displays the number on the screen. I used the JNC (Jump on No Carry) instruction to skip the portion of code to display the number of disk drives in the system. The carry is cleared (no carry) if b0 in the status word is clear.

Conditional transfer instructions are very powerful programming tools. Without them, the microprocessor would be of little value.

Iteration Control Instructions

LOOP, LOOPZ, LOOPNZ

At the beginning of this chapter, I made the statement that the 8088 contains instructions that allow you to perform a function similar to that of BASIC's FOR and NEXT program statements. FOR and NEXT are used in BASIC to repeatedly execute the program statements that appear between the two statements, or within the iterative loop they form. The general form of the FOR and NEXT loop is:

```
10  FOR I = 10 TO 1 STEP -1

20  PRINT "MY NAME IS GARY"

30  NEXT I
```

In the programming example above, line 20 will be executed ten times. That is, MY NAME IS GARY will be printed on the video display ten times. The iteration count is assigned to the variable I and is decremented by 1 each time the NEXT instruction is executed.

In a similar manner, a source program for the 8088 can be created that allows a series of instructions to be repeated a specified number of times. The LOOP instruction is used for this purpose. For example:

```
   LEA  BX,SOME_PART_OF_MEMORY      ;Put address into BX.

   MOV CX,10                        ;Set loop counter to 10.

HERE_IS MOV BX,0                    ;Store zero at word addressed
                                    ;by BX.

   INC BX                           ;Point to next word.

   INC BX                           ;INC twice for words.

   LOOP HERE_IS                     ;Loop until CX = zero.
```

In the programming example above, CX specifies the number of times the loop is to be executed. BX points to the start of a portion of memory we wish to clear. The program clears a word of memory ten times (20 bytes). Each time through the loop, zero is stored at the memory location pointed to the BX register and in the next location (BX + 1). BX is incremented twice, so that it will point to the next word location. The LOOP instruction decrements the counter, CX, by one. If CX does not contain zero, the program loops back to the instruction at the label HERE_IS.

The LOOP instruction can only transfer control to a destination operand within − 128 to + 127 bytes of the LOOP instruction.

There are three variations of the LOOP instruction:

1. LOOP, which decrements the CX register by one and transfers control to the destination operand if CX does not equal zero,
2. LOOPZ (or LOOPE), which decrements CX by one and transfers control if CX is not zero and the zero flag is set, and
3. LOOPNZ (or LOOPNE), which decrements CX by one and transfers control if CX is not zero and ZF is not set (cleared).

Another example of the LOOP instruction is:

```
    MOV   CX,0FFFFH                ;Set up iteration counter.

HERE: IN   AL, TEMP_PORT           ;Input from a port.

    CMP   AL,HOT                   ;Is it equal to HOT?

    LOOPNE    HERE                 ;If not equal and CX not equal
                                   ;to zero, keep trying.

    OUT   CONTROL_PORT,11110000B   ;Turn off warmer.
```

In this example, the program inputs data from the specified port (TEMP_PORT) up to 65,536 times, as designated by the count in CX. If the value obtained from the port is equal to the value assigned to the label HOT or if the CX register is zero, the loop terminates, and the next instruction (OUT CONTROL_PORT,11110000B) is executed.

Jump on CX Equal to Zero

JCXZ

Format: **JCXZ target**

JCXZ (Jump on CX equal to Zero) transfers program control to the target operand if the CX register is equal to zero. For example:

```
MY_LOOP JCXZ  OUT_OF_LOOP         ;If CX is 0, do not execute the loop.

MY_LOOP1 IN     AL,PTR_PORT       ;Get data from port.

        OR      AL,AL             ;Set flags.

        LOOPZ   MY_LOOP1          ;If data from PTR_PORT is zero.
                                  ;Get another byte.

OUT_OF_LOOP   NIL                 ;Program winds up here.
```

If CX equals zero on entry to the routine, control is passed to the program label OUT_OF_LOOP. The loop is not executed.

Interrupts

The three interrupt instructions are used to transfer control to another part of the program. Interrupts can be externally generated, or they can occur via the INT or INTO instructions under program control. When the 8088 encounters one of the two possible software generated interrupts, the 8088 pushes the flag register onto the stack (similar to PUSHF), clears IF and TF in the flag register, and fetches a double word entry in an interrupt vector table stored in memory at 00000H to 003FFH.

There are 256 different possible interrupt types, each with its own entry in the table. Once the 8088 has cleared IF and TF, the stack pointer is decremented by 2 and the current value of the code segment register is saved on the stack. Next, the 8088 calculates the absolute memory address of where the segment vector is stored. The interrupt type (0–255), is multiplied by 4 and added to 2 (TYPE * 4 + 2). This is the segment address of the interrupt service routine. The word stored at this location is then placed into CS.

In a similar manner, the stack pointer is again decremented by two, and IP is saved on the stack. The offset vector address is calculated by multiplying the interrupt TYPE by four. The word stored at this location is then placed into IP and becomes the new segment offset. The next instruction to be executed is specified by the combination of the code segment register (CS) and the instruction pointer (IP).

INT

Format: **INT interrupt type**

INT (INTerrupt) is one of the more powerful instructions the 8088 is capable of executing. With it, you are able to invoke an interrupt service routine specified by the type operand coded into the instruction. For example:

```
INT       21H            ;Perform a type 33 interrupt.
```

As you'll discover in the following chapters, this interrupt type is used by MS-DOS to handle different system services, such as sending a character to the screen.

Now direct your attention to Table 4-4. Notice that there are five reserved and defined interrupts listed in the table. They are:

Type 0 - Reserved for division by zero

Type 1 - Reserved for single stepping

Type 2 - Reserved for Non-Maskable Interrupts (NMI)

Type 3 - Reserved for 1-byte INT instruction

Type 4 - Reserved for signed overflow interrupt (INTO)

Table 4-4
Intel 8088 Interrupt Vectors

Interrupt Type	Memory Location (Absolute)		Definition
0	0000	(IP)	Divide Error
	0002	(CS)	
1	0004	(IP)	Single Step (TF)
	0006	(CS)	
2	0008	(IP)	Non-Maskable Interrupt
	000A	(CS)	
3	000C	(IP)	Breakpoint Trap
	000E	(CS)	
4	0010	(IP)	Overflow Trap
	0012	(CS)	
***********	NOTE! (Types 5 -31 reserved by Intel)		***********
5	0014		Screen Print
6	0018		
7	001C		
**	NOTE! Types 08H through 0FH are generated by the 8259 Interrupt controller		**
08H	0020		Timer (8253)
09H	0024		Keyboard

Table 4-4
Intel 8088 Interrupt Vectors

Interrupt Type	Memory Location (Absolute)	Definition
0AH	0028	Color Board
0BH	002C	N/A
0CH	0030	Serial Adapter
0DH	0034	N/A
0EH	0038	Disk
0FH	003C	Printer
****************	BIOS Interrupts	****************
10H	0040	Video (BIOS)
11H	0044	Equipment Flag (BIOS)
12H	0048	Memory Size Check
13H	004C	Disk I/O
14H	0050	Serial I/O
15H	0054	Cassette I/O
16H	0058	Keyboard I/O
17H	005C	Printer I/O
18H	0060	BASIC
19H	0064	Bootstrap
1AH	0068	Time of Day
1BH	006C	Keyboard Break
1CH	0070	Timer Tick
1DH	0074	Video Initialization
1EH	0078	Diskette Parameters
1FH	007C	Video Graphics
***************	MSDOS Interrupts	***************
20H	0080	Program Terminate
21H	0084	DOS Function Call
22H	008C	Terminate Address
23H	0090	Fatal Error Vector
24H	0094	Absolute Disk Read
25H	0098	Absolute Disk Write
26H	009C	Terminate, Stay Resident
27H-5FH	0100-017F	Reserved
60H-67H	0180-019F	User Interrupts
68H-7FH	01A0-01FF	Not Used
80H-0F0H	0200-03C3	BASIC
0F1H-0FFH	03C4-03FF	Not Used

Table 4-4 Notice Intel reserves interrupts 5-31 (05H - 20H) for future use. The 80188/86, and 80286 are newer and more powerful processors from Intel which use the vectors which are reserved on the 8088. However, IBM decided to use these interrupt vectors for processing many of the BIOS (Basic Input/Output System) functions in ROM in the PC.

Intel reserves interrupt types 5–31 for future use. IBM chose to use these interrupts on the PC. This is one possible source of incompatibility when comparing a compatible computer to an IBM PC.

INTO

Format: **INTO**

INTO (INTerrupt on Overflow) generates a type 4 interrupt if the overflow flag is set in the status register. If OF is not set, the instruction following the INTO instruction is executed. It is used to handle an overflow condition resulting from arithmetic operations. For example:

```
SBB    AX,BX            ;Subtract with borrow BX from AX

INTO                    ;Interrupt if overflow.
```

causes an interrupt type 4 to be generated if, as a result of the subtraction, the overflow flag is set.

IRET

Format: **IRET**

IRET, or Interrupt RETurn, terminates every interrupt service routine. The instruction passes control back to the main program by popping IP, CS, and the flags from the stack. These values are automatically pushed on the stack anytime the 8088 receives an interrupt.

String Manipulation Instructions

String instructions manipulate blocks of data stored in memory. They require that the destination operand be pointed to by DI and that the source pointer be in SI. If the direction flag (DF) is set, then DI and SI are automatically decremented after the string operation has been executed. If the direction flag is clear, the pointers are incremented after the string instruction has been executed.

Due to the use of the index pointers SI and DI and the association they have with DS and ES, it is implied that the source string resides in the data segment (DS:SI) and that the destination string resides in the extra segment (ES:DI). If there was no flexibility allowed in the addressing of operands, these otherwise powerful instructions would become cumbersome and awkward to use.

There is a way to overcome this otherwise serious limitation in operand string addressing. This allows the addressing of source and destination strings that are both in either the data segment or the extra segment. More on this in a moment.

Operands must be specified in the instruction so that the assembler can determine what type of string instruction it is to generate, byte or word. The assembler determines this from the TYPE attribute assigned to each operand when it is defined. For example, MY_DATA_STRING DB 0FFH assigns MY_DATA_STRING a type attribute of BYTE. If the DW pseudo-op were used to define the string, then the type attribute would be WORD.

Based on this information, the assembler generates either a MOVSB or MOVSW instruction. Simply loading the index registers with the address offset of the data referenced is not enough, however. The assembler cannot determine the type attribute from the index registers; therefore, the operands must be included in the instructions when using the generalized form MOVS, CMPS, SCAS, LODS, and STOS.

If you use the short forms of the instructions, such as, MOVSB, MOVSW, CMPSB, CMPSW, SCASB, SCASW, LODSB, or LODSW, the operands do not have to be included in the source statement. The assembler is able to generate the proper instruction, as the data type is specified in the source statement.

Prefix Operators

REP, REPZ, REPNZ

A prefix operator may be specified in the instruction, and it is used when you want to repeat the string instruction a certain number of times. The instruction is repeated until the contents of CX meet the specified criteria. For example, the REP operator is used when repeating an operation until CX equals zero. REPZ causes the operation to be repeated as long as the count in CX does not equal zero and ZF is set to one. It is similar to the LOOPZ instruction previously discussed. REPNZ repeats the specified operation as long as the count in CX is not zero and ZF is clear (zero).

Move String

MOVS, MOVSB, MOVSW

Format: **MOVS destination-string, source-string**
MOVSB
MOVSW

MOVS moves a byte or word from the source string to the destination string pointed to by SI and DI. MOVS is used to transfer a block of data from one portion of memory to another. Although you can override the segment (DS) pointed to by the source operand SI during a repeat operation, Intel cautions you not to do this: "....avoid using the other two prefix bytes with a repeat-prefixed string instruction. One overrides the default segment addressing for the SI operand and one locks the bus masters. Execution of the repeated string operation will not resume properly following an interrupt if more than one prefix is present preceeding the string primitive."

```
MOV   CX,1024                     ;Set up counter for 1K block move
STD                               ;Set the direction flag
LEA  SI,MY_DATA_STRING            ;Get source address
LEA  DI,ES:DI,MY_DATA_STRING2     ;Get destination address
REP MOVS [DI],ES:[SI]
```

The instruction above can be used to move data from one part of the extra segment to another part of the same segment. However, the instruction is not guaranteed to operate properly on returning from an interrupt. What can you use to move data from one location within the same segment (ES)? Try this:

```
ASSUME DS:EXTRA_SEGMENT,ES:EXTRA_SEGMENT
MOV  AX,EXTRA_SEGMENT           ;Establish seg. registers
MOV  DS,AX                      ;Set up DS register
MOV  ES,AX                      ;Set up ES register
MOV CX,COUNT                    ;Load repeat counter
STD                             ;Set direction flag
LEA SI,MY_DATA_STRING           ;Load source string DS:SI
LEA DI,ES:MY_DATA_STRING2       ;Load destination string ES:DI
REP MOVS [DI],[SI]              ;Move data from extra seg. to extra seg.
```

By using the ASSUME statement, both DS and ES are set to the start of the extra segment. When the repeated move string instruction is encountered, data is moved from one part of the extra segment to the other. You can also use this programming quickie to move data from one part of the data segment to another. Simply set both DS and ES to the start of the data segment rather than using the extra segment.

Another alternative is to use the LOOP instruction to transfer a block of data from one part of the extra segment to another part of the extra segment. Using the LOOP instruction allows you to use the segment override operator with the source operand. For example:

```
   LEA  DI,ES:MY_STRING               ;Set up destination string pointer

   LEA SI,MY_STRING2                  ;Set up source string pointer

   STD                                ;Set direction flag

   MOV CX,1024                        ;Set up iteration counter

MOVE_IT MOVS MY_STRING,ES:MY_STRING2  ;Move string

   LOOP MOVE_IT                       ;Repeat until CX = 0
```

This program code is shorter and less complicated than the previous example, but it will not work when both the destination and source strings are in the data segment. You will have to again set ES and DS to point to the start, or base, of the data segment. This involves using the ASSUME statement as demonstrated before.

MOVSB and MOVSW do not require operands, as they specify the data type attribute directly within the source statement.

Compare String

CMPS, CMPSB, CMPSW

Format: **CMPS destination string,**
source string
CMPSB
CMPSW

Often it is necessary to search or scan strings for a matching byte or word value or for a byte or word value that differs between strings. CMPS does exactly this and sets the appropriate flags in the status register.

REPZ and REPNZ can be used to repeat the operation until the condition is met. REPZ continues to compare the source to the destination string as long as CX is not zero and the operands match (ZF = 1). When the operands differ, ZF is cleared, and the next instruction following the CMPS is executed. In most cases, the instruction following the CMPS will be a conditional jump instruction.

REPNZ is used to compare strings for a matching byte or word. The operation is repeated as long as CX is not zero and ZF is cleared.

An example of the CMPS instruction is:

```
    LEA    SI,MY_STRING                  ;Set pointer to source string

    LEA    DI,ES:MY_STRING2              ;Set pointer to destination

    STD                                  ;Set direction flag

    MOV    CX,1024                       ;Set counter to 1024 bytes to
                                         ;compare.

    REPZ CMPS MY_STRING2,MY_STRING       ;Find a byte that differs.

    JZ     GOT_ONE                       ; ZF=`1', then got a match.

    JMP    EXIT                          ;Else leave routine

GOT_ONE: LODS MY_STRING                  ;Load string value and inc. SI
EXIT: RET
```

The compare string instruction updates all the flags of the status register and, like the CMP instruction, affects neither operand. CMPSB (byte compare) and CMPSW (word compare) do not require operands, as the data type operator is specified in the source statement.

Scan String

SCAS SCASB, SCASW

Format: **SCAS destination string**
SCASB
SCASW

The SCAn String instruction compares AL (byte) or AX (word) to the destination string. If you want to search memory for the ASCII letters IBM, use the SCAS instruction as follows:

```
    LEA  DI,ES:MY_STRING               ;Set up destination string
                                       ;Assume Memory defined as words

    MOV CX,1024                        ;Search 1K of memory

    STD                                ;Scan backwards
    MOV AH,"."                         ;Set up value to search for

    MOV AL,"M"                         ;

SCAN_MEM: REPZ SCAS MY_STRING          ;Scan for a match
                                       ;Continue if there is none

    JZ GOT_MATCH_1                     ;Got it, good

    JMP EXIT                           ;Otherwise stop searching

GOT_MATCH_ONE MOV AH,"B"               ;Next 2 bytes must match

    MOV AL,"I"                         ;value in AX

    SCAS MY_STRING                     ;Next word match?

    JZ GOT_IT_ALL                      ;Yes jump to new part of program

EXIT: RET                               ;No, then leave routine.
```

The memory scan terminates if CX=0 or a match is not found in the loop. It also terminates if the second scan fails to find the letters IB.

You can also use the implicit forms of SCAS; they are SCASB (scan for byte) and SCASW (scan for word). These instructions do not require an operand, as the data type of the destination operand pointed to by DI is specified in the instruction. The assembler is able to generate the proper instruction (byte or word scan), as the data type attribute is specified in the source statement.

DI is updated after the execution of the SCAS instruction to point to the next data element. If DF is clear, DI is incremented. If DF is set, DI is decremented, as in the example above.

Load String

LODS, LODSB, LODSW

Format: **LODS source-string**
LODSB
LODSW

LODS (LOaD String) transfers a byte into AL or a word into AX. SI must point to the source string, and it is updated after the instruction's execution to point to the next data element. Although you could specify a repeat prefix as part of the instruction, this is not usually done. Since AL would only retain the last value fetched from memory, repeating the operation makes little sense. It is useful, however, when you need to load a value from memory and automatically update the pointer.

An example of the LODS instruction is:

```
   LEA  SI,MY_STRING                ;Set up source pointer

   LEA DI,ES:MYSTRING2              ;Set up destination string

   STD                              ;Set direction flag

   REPZ CMPS MY_STRING2,MY_STRING   ;Compare strings for
                                    ;difference

   JNZ FOUND_ONE                    ;Found one

EXIT: RET                           ;None found return

FOUND_ONE: LODS MY_STRING           ;Get the element that differs STOS
```

Store String

STOS, STOSB, STOSW

Format: **STOS destination string**
STOSB
STOSW

The STOre String instruction transfers a byte from AL or a word form AX to the destination operand pointed to by DI. STOS is most frequently used in initializing data areas to a constant value. Often it is necessary to clear a data area prior to using it in a program. STOS provides a convenient way of doing just that. For example:

```
LEA  DI,ES:MY_STRING_AREA     ;Set pointer to data area

MOV CX,2048                   ;Clear 2048 words (4096 bytes).

OV AX,0                       ;Value to store

REP STOS MY_STRING_DATA       ;Repeat until CX - 0
```

Processor Control Instructions

These instructions govern the 8088 during program execution. They control the microprocessor and affect the manner in which the 8088 executes other instructions.

No Operation

NOP

Format: NOP

The NO oPeration instruction can be used as a time filler or to fill an area in memory which can be used to patch the program when bugs are discovered. When the 8088 encounters the NOP instruction, it does nothing. NOP does not affect any of the flags.

Carry Instructions

CLC, CMC, STC

Format: CLC
CMC
STC

These three instructions affect the carry flag. CLC clears the carry, STC sets the carry, and CMC complements the carry bit (reverses its state). As the instructions implicitly designate the carry bit, they require no operands.

Some examples of their usage are:

EXAMPLE 1:

```
CLC            ;Condition the carry bit for the following
               ;rotate instruction.

RCL            ;Rotate left through carry
               ;the 0 originally in the carry is
               ;now in B0.
```

EXAMPLE 2:

```
STC            ;Set the carry flag to 1

RCR            ;Stuff a 1 into b7 and put b0 in carry.
```

CLD/STD

Format: CLD
STD

These instructions affect the direction flag, DF. CLD is used to clear DF and causes the index register(s) to be automatically incremented during execution of the string instructions. STD is used to set DF and causes the string instructions to automatically decrement the pointer(s). See the previous discussion on string operations for examples of their usage.

Interrupt Instructions

CLI and STI

Format: CLI
STI

CLI (CLear the Interrupt flag) and STI (SeT the Interrupt flag) are used to enable or disable external interrupts. CLI disables external interrupts, while STI enables them. There are maskable and nonmaskable interrupts. Nonmaskable interrupts are always honored; they cannot be disabled by clearing IF.

Masking interrupts is common when you don't want an external event to interrupt the microprocessor. Going back to the human analogy of Chapter 3, using the STI instruction is a little like hanging a sign outside your door, that says "DON'T BUG ME, I'M BUSY!" When you are able to answer the door, you use the CLI instruction, which removes the sign from the door and enables interrupts.

Delay Instructions

HLT and WAIT

Format: HLT
WAIT

Have you ever had to wait for an important message and sat around waiting for the telephone to ring? The WAIT instruction puts the 8088 into an idle or wait state.

One of the pins on the 8088 microprocessor is used as an input signal and is called TEST. As long as this line is inactive, the 8088 waits for either the TEST input to become active or for an external interrupt to occur. No other instructions are executed while the 8088 is waiting.

When the TEST input becomes active (logic 0, or 0 volts), the 8088 executes the instruction following the original WAIT instruction. Should an external interrupt occur, the 8088 saves the address of the WAIT instruction on the stack and vectors to the appropriate interrupt service routine. When the interrupt has been serviced, the address of the WAIT instruction is popped from the stack, and the 8088 reenters the wait state.

The HLT (HaLT) instruction is similar to the WAIT instruction, in that they both put the 8088 into an idle condition. But unlike the WAIT instruction, HLT does not sample the TEST input. When halted, the 8088 waits for a reset, or an enabled maskable or nonmaskable interrupt. Once the interrupt occurs, the halt state is cleared. Both the HLT and WAIT instructions are used to synchronize the 8088 to external events.

Instructions to Other Processors

ESC and LOCK

Format: **ESC external op-code,source**
LOCK op-code destination operand,source operand

ESC (ESCape) is used to send instructions to other processors, such as the 8087 numeric co-processor. When the 8088 encounters the ESC instruction, the processor executes a NOP instruction. The only other operation that the 8088 performs is to address the source operand specified. The external op-code must be a 6-bit number and is composed of 3 bits indicating the co-processor that the instruction is intended for and 3 bits designating the instruction the co-processor is to execute. If the operand is in memory, the 8088 reads its value and ignores it. The co-processor, however, also reads the value and uses it as the source operand in its operation.

LOCK is not an instruction, but rather a 1-byte prefix to an instruction. It is used to LOCK other processors out of the system for the duration of the instruction. When the 8088 encounters the LOCK prefix byte, it asserts a signal on the LOCK pin of the microprocessor. The LOCK signal remains active during the execution of the instruction.

It is common when two or more processors share resources, such as memory, that RAM semaphores are used to communicate between them. LOCK prevents another processor from inadvertently changing a semaphore while the semaphore is also being altered or read by the first process.

Imagine what would happen if both you and your spouse were to make a major purchase. Your spouse has just purchased a new car for your birthday present but has not changed the balance shown in your bankbook. It so happens that you too have made a major purchase, a trip for two on a Caribbean cruise. Unfortunately, the combined cost of both purchases is more than what you have in savings. Perhaps what you need is a LOCK on your savings book that would have prevented one of you from making a major purchase until the balance was recalculated and written in the checkbook. The LOCK prefix does exactly that.

Chapter 4 Review

1. Write the necessary program statement(s) to add 20 to the memory location COUNTER.
2. There are ____________ major groups of 8088 instructions.
3. Write the program statements that will compare a value in the memory location COUNTER to 20 and branch to the program label THERE if the value in COUNTER is equal to 20.
4. What does this instruction sequence do?

   XOR AX,AX
5. What is wrong with this statement:

   MOV 10H,AL
6. What is wrong with this statement:

   MOV FACEH,AX
7. For every PUSH there must be a ____________.
8. Write a program loop that will clear 64 bytes of memory. Begin clearing memory at the label DATA_STRC.
9. Write two statements that will load the offset address of COUNTER into the register BX.
10. List two variations of the LOOP instruction. What does each do when executed?

5

Assembler Features

Until now, I have refrained from using all but the minimum of assembler features in the programming examples. Although the IBM Macro Assembler manual provides an overview of the directives and pseudo-ops available within the assembler, the descriptions often lack adequate examples of their usage. In fact, the manual starts out with the statement, "This manual is a reference manual for experienced assembler language programmers, like yourself, who use the IBM Personal Computer Macro Assembler." If you're not an experienced assembler language programmer, you must have wondered to whom IBM was referring.

Therefore, I want to not only discuss the features available within the assembler but also give several examples of HOW the features can be used to save you time in your programming efforts.

You do not have to be an experienced programmer to understand this chapter. Although this chapter stands on its own, it will be helpful to reread the IBM Macro Assembler manual's description of the features discussed. It's not my intention to eliminate the need for the Macro Assembler manual, but merely to provide alternate explanations in an attempt to clarify the many ambiguities that exist within the manual.

Assembler Format

The format for entering your source programs has been given previously, but I will repeat it here. The general form for entering source statements is:

<--NAME--> <--OP-CODE--> <--operand(s)--> <--comments-->

The assembler directives and pseudo-ops that define data or control the assembly process are entered in the op-code field, which is normally reserved for a valid 8088 instruction. Pseudo-ops allow you to define data, associate constant values with labels (symbolic names), and control the assembly. The types of allowable pseudo-ops available within the IBM Macro Assembler fall into four categories: data definition (including symbol, procedure, and segmentation definitions), conditional, macro (discussed in Chapter 6), and listing pseudo-ops.

Because pseudo-ops appear in the source file in the op-code field, it may be confusing at first in differentiating between an 8088 instruction (op-code) and an assembler directive (pseudo-op). When a name, operand, or comment is required with a pseudo-op, the same rules apply to the name field as when a name is required for an op-code (see Chapter 2).

Data Pseudo-ops

The data pseudo-ops (see Table 5-1) are discussed in the IBM Macro Assembler manual beginning on page 5-3. Some of the pseudo-ops may seem self-explanatory, while others are not so obvious.

ASSUME

***Format:* ASSUME seg reg:seg name[,....]**

ASSUME tells the assembler which segment registers are to be associated with each segment. In the general format, seg reg refers to the segment registers CS, DS, ES, or SS. Seg name is the name you specify for the given segment. For example:

```
ASSUME      CS:MY_CODE,DS:MYDATA,ES:NOTHING,SS:MY_STACK
```

Table 5-1
Data Pseudo-ops

Category I **Segment Definitions**	**Category II** **Data Definitions**
ASSUME	DB (Define Byte[s])
SEGMENT	DW (Define Word[s])
ENDS	DD (Define Double word[s])
ORG	DQ (Define Quad-Word[s])
GROUP	DT (Define Ten-Bytes)
EVEN	
Category III **Procedure Related**	**Category IV** **Symbol Definitions**
PROC	EQU
ENDP	=
	LABEL
Category VI **Record/Structure Related**	**Category VII** **Module/Assembler Related**
RECORD	NAME
STRUC	.RADIX
	PUBLIC
	EXTRN
	INCLUDE
	END

informs the assembler to associate the CS register with the segment defined with the label MY_CODE, and DS with the segment labeled MY_DATA. ES is associated with nothing. This means that any previous assumptions about the ES register are canceled. The segment registers can appear in any sequence, and all four need not be present.

You may want to cancel a previous assumption about the DS register; in which case, the statement:

```
ASSUME      DS:NOTHING
```

would be all that is required. Figure 5-1 illustrates how the assembler associates segment, base, and index registers with each of the four possible segments. You can have more than one of the segment types in your program (e.g., two code segments), but only one of each segment type is active at any given time.

Although the assembler has been informed via the ASSUME directive to associate certain segment registers with the various segments in a program, the segment registers themselves must contain the proper base address for the given segment.

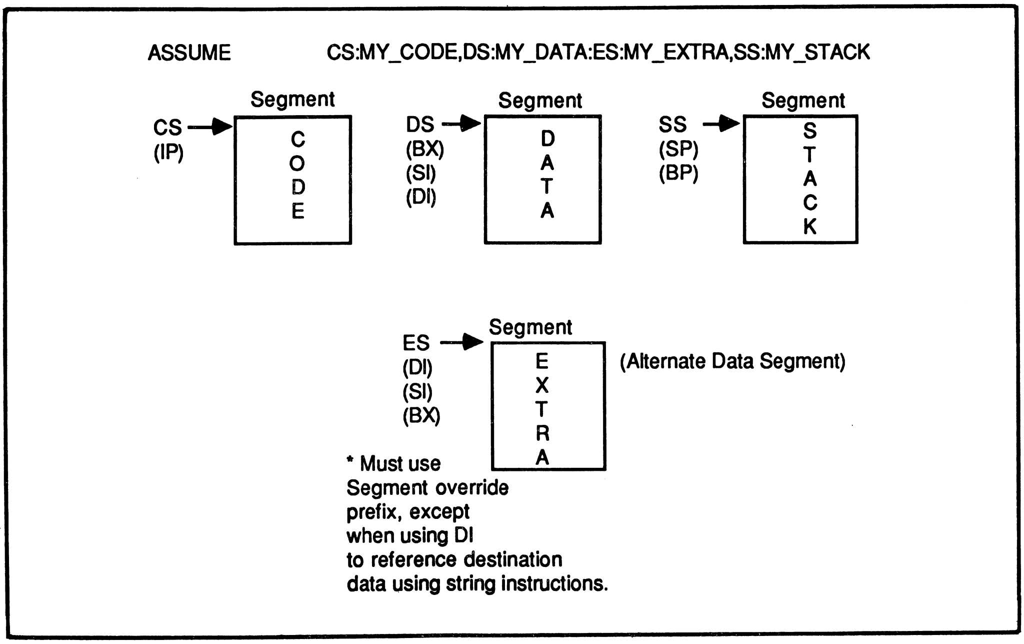

ASSUME CS:MY_CODE,DS:MY_DATA:ES:MY_EXTRA,SS:MY_STACK
Segment
CS
(IP)
CODE
Segment
DS
(BX)
(SI)
(DI)
DATA
Segment
SS
(SP)
(BP)
STACK
Segment
ES
(DI)
(SI)
(BX)
EXTRA
(Alternate Data Segment)
* Must use Segment override prefix, except when using DI to reference destination data using string instructions.

Figure 5-1 ASSUME Directive

Example:

```
ASSUME      CS:MY_CODE, DS:MY_DATA, ES:MY_DATA, SS:MY_STACK

MOV    AX, MY_DATA                  ;Set up segment registers

MOV    DS, AX                       ;Transfer seg-base to DS

MOV    ES, AX                       ;Do the same for ES
```

You should notice that the segment registers cannot be used as destination operands in immediate move instructions. The segment base address must first be moved into a general purpose register (usually AX) and then transferred to the proper segment register. Additionally, CS cannot be used as the destination operand of any move instruction.

SEGMENT

Format: **seg name SEGMENT [align] [combine] [class]**

ENDS

Format: **seg name ENDS**

These pseudo-ops are used to define a segment. There must be a segment for your program code, data, and stack. Since there are two data segment registers, DS and ES, your program can contain two different active data segments at a given time. The programming example in Listing 2-1 contains a data segment, a stack segment, and a code segment.

Use the SEGMENT and ENDS pseudo-ops in the following manner to define your segments:

```
MY_STACK   SEGMENT   PARA STACK    'STACK'

      DW   100 DUP(?)

MY_STACK   ENDS

MY_DATA    SEGMENT   PARA 'DATA'   ;Define data segment

TABLE1     DW   100 DUP(0)         ;Define 200 bytes of zeros

   ..                              {More data definitions}

   ..

   ..

MY_DATA    ENDS

MY_CODE    SEGMENT    PARA 'CODE'  ;Define code segment

  ASSUME   CS:MY_CODE,DS:MY_DATA,SS:MY_STACK

     ..

     ..

     ..

MY_CODE    ENDS
```

In the preceding example, the code segment is defined to have an align type of PARA, which aligns the segment on a boundary address that is divisible by 16. The class type must be enclosed in single quotation marks and is used by the linker to combine segments with identical class names. The stack segment uses the combine type, which in this case is STACK. The linker requires there to be a stack segment defined when the combine operator is present. See Chapter 5 in the IBM Macro Assembler manual for a full description of the optional align, combine, and class operands.

PROC

Format: **procedure name**
PROC [NEAR]
or
procedure name PROC FAR

ENDP

Format: procedure name ENDP

The concept of modular programming is based upon building small, unique blocks of code. Each block of code performs a specific task. Modularity is beneficial for several reasons. Smaller chunks of code are easier to debug than a large program. The code is also easier to understand. Procedures reduce the amount of code required in a given program by generating the object code required for a specific function once and allowing the code to be used by other parts of the program.

PROC defines a section of code as a procedure. Old-timers to ALP will recognize that procedures are what used to be called subroutines. A RET (return) instruction returns control to the calling program.

If the PROC is defined—as with a distance attribute of NEAR (within the same segment as the calling program), then the assembler generates a RET instruction that restores the offset contained in IP before the procedure is called. If the procedure is defined as being of the FAR type, then the RET instruction restores both IP and CS to the segment offset and segment of the calling program. NEAR is used for intrasegment calls, and FAR is used for intersegment calls.

Example:

```
MY_PROC     PROC NEAR           ;Callable from same segment.

            ..

            ..

            RET

MY_PROC     ENDP                ;End of procedure.

    PUBLIC       DIFF_SEG_PROCEDURE

DIFF_SEG_PROCEDURE      FAR     ;Procedure to be called by other

                                ;segments

            ..

            ..

            RET

DIFF_SEG_PROCEDURE      ENDP
```

Notice that the procedure must not only be declared FAR, but it must also be declared PUBLIC. This allows other programs not defined in the module to reference the procedure. ENDP terminates the portion of code and the definition of the procedure.

Defining Data

DB, DD, DW, DQ, DT

Format: **name Dx expression**
(Where x is either
B, D, W, Q, or T)

These pseudo-ops allow you to define a variable or initialize storage locations. The pseudo-ops may be preceded by a symbolic name and must be followed by an expression. The second letter of the designation defines the TYPE attribute associated with the storage location(s). For example:

```
ONE_BYTE   DB ?
```

defines 1 byte of storage. The ? indicates that the byte is not initialized to any particular value. You can also define more than one byte of storage as in:

```
DECIMAL_TABLE   DW      10000,1000,100,10
```

These pseudo-ops are used to establish data areas and tables necessary for program execution, character code translations, etc. In Listing 2-1, another use for the DB pseudo-op was demonstrated: the creation of ASCII message strings. For example:

```
MESSAGE_ONE      DB    'This is the first message I want to define.' ,

                 DB    '$'
```

In this example, the string appears in memory exactly as it appears between the single quotation marks. The second DB defines a byte containing 24H, the ASCII code for the dollar sign.

Similarly, DW, DD, DQ, and DT define words, double words (4 bytes), quad words (8 bytes), and 10 bytes of storage.

DUP

An operator which may be used with any of the data definition pseudo-ops is the DUP (duplicate) operator. The DUP is used to create or initialize data storage. For example:

```
DW   100  DUP(?)
```

defines 200 bytes of uninitialized storage. You can specify any valid initialization value in the DUP clause. For example:

```
DW    100   DUP(0)
```

initializes 100 words of storage to 00H. The IBM Macro Assembler manual discusses each of the data definition pseudo-ops in detail.

Symbolic Names

Attributes and Operators

Symbolic names carry with them more than what you put into them. You may choose a symbolic name that really describes a function of the code where the label is placed. For example:

```
FIND_SQUARES:
```

may do just that, find the square of a number. However, the assembler assigns its own meaning to the label (it really doesn't know that the statements following the label actually do find the square of a number). The meanings the assembler assigns to the labels are known as attributes. They describe how the label is used.

Segment Attribute

Each label has a segment attribute associated with it. It is the value of the beginning of the segment where the label is defined. For example:

```
MY_DATA     SEGMENT     PARA 'DATA'

ASCII       DB     128 DUP(?)

MY_DATA     ENDS
```

assigns the segment attribute of ASCII the value of MY_DATA. The segment register (DS, in this example) must contain the segment value in order to access the label ASCII. In order to find the segment value of ASCII, you would use the value-returning operator SEG:

```
MOV    AX,   SEG ASCII          ;Get segment value of ASCII

MOV    DS,AX                    ;Set up segment register
```

Offset Attribute and Operator

By now you should know that whenever there is a segment, there is an offset into the segment. An offset attribute is associated with every label in a program. The offset for ASCII in the above example is zero. The first byte of data defined by the label ASCII is zero bytes from the beginning of the segment where it is defined. The offset attribute is a 16-bit unsigned value. You can use the value-returning operator OFFSET to load the offset value into a register:

```
MOV   DI,OFFSET ASCII        ;Get offset of label ASCII
```

Remember, this instruction moves the offset from the beginning of the segment where the label is defined, into the specified register. It does not move the contents of the memory word into the register.

Type Attribute and Operator

Along with a segment and offset attribute associated with a name, the assembler associates a type attribute with the symbolic representation. This attribute defines the number of bytes reserved for the symbol. If a symbol is defined as follows:

```
COUNTER   DW   ?             ;The TYPE attribute is word.
```

The DW pseudo-op is used to Define a Word of storage for the label COUNTER. Therefore, the type attribute is word (2 bytes).

In using the instruction:

```
MOV   AX,TYPE   COUNTER
```

the value of 0002H is returned to AX.

There is also a distance attribute associated with memory locations. It informs the assembler whether the memory location is defined as a NEAR or FAR type. For example:

```
SQUARES:   MOV AX,VALUE        ;Get number to square
```

defines SQUARES as having a NEAR attribute. The type operator can be used to return the value (–1 for NEAR and –2 for FAR attributes) based on the distance attribute of the symbol.

Example:

```
MOV   AX,TYPE   SQUARES        ;Will return a -1
```

The possible type values are given in Table 5-2.

Table 5-2
Values Returned by TYPE Operator

TYPE	VALUE RETURNED
DB	1
DW	2
DD	4
DQ	8
DT	10
NEAR	-1
FAR	-2

LENGTH and SIZE Operators

The value-returning operators LENGTH and SIZE are used in conjunction with the type operator. If you define a table of data as:

```
MY_TABLE     DW     10 DUP(?)     ;
```

the LENGTH operator returns the value used to specify the number of times you want to duplicate the word. The SIZE operator returns the size of the data type as LENGTH × TYPE. Since the variable in this example is a word, the type will be 2. The operator length is 10; therefore, the size of the variable will be equal to LENGTH × TYPE = 10 × 2 = 20 bytes.

These operators are useful in moving the LENGTH and SIZE values into the count register (CX) prior to an iterative loop. For example:

```
MOV    CX,LENGTH MY_TABLE
```

moves the length of MY_TABLE into the CX register. In this instance, the LENGTH operator returns the total number of word entries in the table, which is 10.

In a similar manner, the SIZE operator returns the total number of bytes allocated to the variable. For example:

```
MOV    CX,SIZE     MY_TABLE
```

returns the value 0014H (20 decimal) to the CX register. LENGTH and SIZE have relevance only when the variable has been duplicated (DUP).

More Pseudo-ops

EQU

Format: name EQU expression

The equate pseudo-op assigns the value in the expression field to the symbolic name that precedes the EQU pseudo-op. Some examples of the use of EQU are as follows:

```
THOUS       EQU   1000         ;THOUS equated to 1000

NEG_VAL     EQU   -5000        ;NEG_VAL equated to minus value

COMP_VAL    EQU   THOUS*5      ;Computed as 5 thousand

COUNT_REG   EQU   CX           ;Represent the CX register
```

Notice that you can use arithmetic operators within the statement and let the assembler compute the values. I will discuss arithmetic operators later in this chapter. The name used with the equate statement is mandatory and must not be followed by a colon.

You can also use a label to represent another label or a register. In the examples above, COUNT_REG is equated with register CX. If you use labels to represent the registers of the 8088, define them prior to using them in your program. If you do not define them prior to their usage, the assembler will generate an error. Labels that are found to be names for the 8088's registers cannot be forward referenced.

=

Format: name = expression

The = (equal) pseudo-op works in a manner similar to the EQU pseudo-op with one major exception; equal allows you to reassign the symbolic name to a new value. You cannot do this with the EQU pseudo-op.

Assume for a moment that you want to assign different values to the name MAIN_COUNT. The constant will be used to load a count value in one of several different delay routines. You could use the = pseudo-op to redefine the label within the program. Often this is done at the beginning of a module that needs to use a label, with a different value than the one previously assigned.

Examples:

```
MAIN_COUNT      =     1000     ;Value of MAIN_COUNT = 1000

MAIN_COUNT      =     10000    ;MAIN_COUNT is redefined.
```

LABEL

Format: name LABEL type

The LABEL pseudo-op assigns the segment, offset, and type attributes of name. Examine the following portion of code:

```
ARRAY_BYTE        LABEL       BYTE          ;ARRAY_BYTE = Current segment
                                            ;with the offset at current
                                            ;byte, and the type attribute
                                            ;of byte.

ARRAY_1     DW     200 DUP(?)               ;This array has 400 bytes
                                            ;(200 words).
```

Notice that ARRAY_BYTE is defined just prior to ARRAY_1. Therefore, the segment is the same for both names, as is the offset of the first byte. However, the array can now be accessed in either of two ways, by byte or word. We already know that the statement:

```
MOV   AL,ARRAY_1
```

generates an error, because the source and destination are not the same size (byte/word). However, by using the LABEL pseudo-op in the manner shown above, you can access a byte within the array, as in:

```
MOV    AL,ARRAY_BYTE
```

You can also use the LABEL pseudo-op to define an entry point in your program as FAR, which would allow other segments to call or jump to that portion of code.

Example:

```
PUBLIC       GET_CHARF

GET_CHARF        LABEL       FAR             ;Define as FAR

GET_CHAR:        IN      AL,DATA_PORT        ;NEAR attribute
```

allows the routine to be called either by code within the segment (NEAR calls and jumps) or by code in a different segment (FAR calls and jumps).

EXTRN

Format: **EXTERN name:type[,...]**

PUBLIC

Format: **PUBLIC symbol**

These pseudo-ops declare symbols and names used in the present module as being defined either in another module (EXTRN) or in this module and accessible by others (PUBLIC). This again points to the use of structured and modular software. You may need to use routines or data which have been defined in other modules; EXTRN and PUBLIC allow you to do this. Both modules will ultimately reside in the same segment.

Example:

```
MY_CODE1      SEGMENT

      PUBLIC       SINE          ;Sine routine defined as public.

SINE:    MOV    AL,VALUE         ;Routine to find the sine of an angle.

            ..

            ..

            RET

MY_CODE1    ENDS

MY_CODE1    SEGMENT

      EXTERN       SINE:NEAR     ;The name 'SINE' is not
                                 ;defined in this module.

            ..

            ..

            CALL SINE            ;In other module, same segment.

            ..

            ..

MY_CODE1    ENDS
```

In the example above, the routine to find the sine of a given value is in another module. Therefore, the label is declared PUBLIC in the module where it is defined and external (EXTRN) in the module that calls the routine.

COMMENT

Format: **COMMENT delimiter text delimiter**

In previous chapters, I have always used a leading semicolon prior to a comment line. That is one method you could use to enter comments into a source program. However, there is another method you can use to comment your programs. Until this chapter, all comments have been entered as follows:

```
MOV     AX,MY_DATA              ;Get the segment start address

MOV     DS,AX                   ;Initialize the data segment
                                ;register.
```

The IBM Macro assembler allows you to enter a comment block without beginning each comment line with a semicolon. The COMMENT pseudo-op can be used as follows:

```
COMMENT * You can enter a block of comment lines

(as many as you like), after the delimiter

character, which is the first nonblank character

following the pseudo-op. The block must end with

the same character as that which opened the

block (asterisk). Use this pseudo-op to enter

a description of the program. *
```

You can choose any character you want as the delimiter character. It is a convenient way to document your programs.

ORG

Format: **ORG expression**

This pseudo-op sets the assembler's location counter to the value specified. At times it is necessary to specify absolutely where a portion of code or data is to reside (as in PROM resident data). ORG lets you specify where the code will be assembled in memory. The use of the ORG pseudo-op forces the assembler to create an absolute (nonrelocatable) object deck.

When the $ (dollar sign) is used with ORG, the current value of the location counter is referenced. You can use this feature to assemble code at some specified offset from the current value of the location counter. For example:

```
ORG     $+0AH
```

sets the location counter to the tenth byte from the current value. Figure 5-2 illustrates this concept further. Other examples of the use of ORG are:

```
ORG     01F0H          ;Sets location counter to 01F0H

ORG     0FF0H          ;Sets the location counter to 0FF0H
```

```
                ORG  100H                  ;Set assembler location
                                           ;counter to 0100H
                DB   20H DUP (41H)         ;Define 32 bytes
                ORG  $+40H                 ;Skip 64 bytes and start
                                           ;at 120H + 40H = 160H
                DB   20H DUP (42H)         ;Define 32 more bytes

                   MEMORY
100H ---->          41H      Which is ASCII 'A'
101H ---->          41H
 |                   |
11FH ---->          41H      End of first DB data area
120H ---->           ?       Not defined
121H ---->           ?
 |                   |
 |                   |
15FH                 ?
160H ---->          42H      Which is ASCII 'B'
161H ---->          42H
 |                   |
 |                   |
17F ---->           42H      End of 2nd DB data area
180H ---->           ?       Not defined
```

Figure 5-2 ORG Directive

The ORG statement is used to manipulate the assembler's location counter. In Figure 5-2, a 32-byte data area is defined, starting at location 100H. The next data area is set to the current value of the location counter plus 40H (160H). The locations from 120H to 15FH are not defined or initialized.

You cannot use the ORG pseudo-op within a STRUC-ENDS definition.

EVEN

Format: **EVEN**
(No operands)

EVEN forces the assembler to align the location counter on an even address. This is most often used when creating programs destined for the 8086 processor, which will fetch data from memory faster if the data is aligned to an even address.

If the location counter is at an odd location, EVEN causes the assembler to generate a 1-byte NOP instruction and increment the location counter to the next address location; otherwise, EVEN does nothing.

RECORD

Format: **name RECORD field**
name:width[=exp],[....]

RECORD allows the definition of bit patterns used for byte or word bit packing. RECORD creates a template for the specified bit pattern; however, you must still allocate storage within a data segment for the pattern's storage.

The pattern can be from 1 to 16 bits in length. Any RECORD containing a bit pattern of 8 bits or fewer will require 1 full byte of storage. A bit pattern of 9 to 16 bits will use 2 bytes, or a word.

Once the RECORD has been defined, storage must be allocated. The general form is:

```
[name] recordname [exp] [,....]

or [name] recordname exp DUP([exp])
```

To allocate storage for the RECORD within the data segment use statements similar to the following:

1. Specify your record at the beginning of your program:

```
MY_BITS    RECORD    RECA:4,RECB:4,RECC:4,RECD:4    ;Record template
```

2. Allocate storage for the record in the data segment:

```
MY_DATA      SEGMENT    PARA 'DATA'

ERROR_BITS   MY_BITS  (0000B,0011B,1100B,1111B)    ;Define bit
                                                    ;patterns
                                                    ;and storage.

     ..

     ..

MY_DATA      ENDS
```

Here a 16-bit pattern of 0000001111001111B is defined. Each field within the record in this example is 4 bits wide, and there are 4 fields (RECA-RECD) total. Notice that the record name becomes the op-code for storage allocation.

The operators WIDTH and MASK are used to access and manipulate data within the record. For example:

```
MOV    CL,WIDTH RECA
```

moves the width of the field RECA (4) into the CL register. You can use the WIDTH operator to move the width of the entire record into a register by using the record name rather than the field name as in:

```
MOV    AL,WIDTH MY_BITS          ;Moves 16 (10H) into AL.
```

MASK returns a field of 1's for the width of the specified field. In other words:

```
MOV    BX,MASK    RECC
```

moves 0000000011000000B to the BX register, and

```
MOV    BX,MASK    RECD
```

returns 0000000000001111B to the BX register.

If you want to move a field into a register and right-justify the bit pattern for the field, first use the field name to obtain the shift count. The shift count is the number of bit positions the specified bit pattern must be shifted to right-justify the pattern.

Example:

```
MOV    CL,RECB              ;GET SHIFT COUNT (8)

MOV    BX,ERROR_BITS        ;GET ALL BITS IN RECORD
                            ;0000001111001111B

SHR    BX,CL                ;SHIFT 8 BITS.
                            ;0000000000000011B
```

The only important point to keep in mind is that shift counts are almost always loaded into the CL register. When used in a shift or rotate operation, the CL register specifies the number of bits to shift the operand (BX, in the example above).

STRUC

ENDS

Format: **structure name STRUC**
...
variable name (data definition pseudo-op)
...
structure name ENDS

The STRUC pseudo-op operates in a manner similar to RECORD but allows the definition of storage at the byte level rather than the bit level. By using the data definition pseudo-ops DB, DW, DD, DQ, and DT, data is defined within the structure. Variable names used to define the data can then be used to access data within the structure.

Storage must be allocated in a manner similar to the RECORD pseudo-op. However, data that is allocated or initialized as single entries can be overridden with the allocation statement. Multiple entries cannot be overridden. See the examples in the *IBM Macro Assembler* manual on page 5-43 for more details on overridable and non-overridable entries.

You must reference data in the structure in the following manner:

1. Define the structure at the beginning of your program:

```
AREA STRUC

WIDTH1      DW      ?               ;STORAGE FOR WIDTH

LENGTH      DW      ?               ;STORAGE FOR LENGTH

AREA        ENDS                    ;END OF STRUCTURE
```

2. Allocate storage within the data segment:

```
MY_DATA     SEGMENT     PARA 'DATA'

AREA_DATA AREA (25,30)

     ..

MY_DATA     ENDS        ;End of the structure
```

3. To reference the data, use statements similar to:

```
MOV    AX,AREA_DATA.WIDTH1
```

NAME

Format: NAME module name

This pseudo-op allows you to assign a unique name to a program module. It is usually used at the beginning of the actual program, prior to the actual program code. You cannot use a reserved word (8088 instructions or pseudo-ops) as the module name. The linker uses either the name assigned to the module or the first 6 valid characters of the TITLE directive if the NAME pseudo-op is not used. If there is no TITLE directive or the characters in the directive are invalid, the source filename is used as the module name.

Example:

```
NAME MY_PROGRAM          ;Assigns the name of MY_PROGRAM
                         ;to the module.
```

INCLUDE

Format: filename

The INCLUDE statement causes the assembler to include the source code from the specified file in the current assembly. The filename must be a valid MS-DOS file specification, such as B:MATHFLE.ASM. The source statements are read from the file MATHFLE.ASM and assembled into the current source code, beginning at the INCLUDE statement.

The IBM Macro Assembler manual states that "nested INCLUDEs are not allowed. If they are encountered, then an error message results." A nested INCLUDE occurs when you use the INCLUDE directive and the file specified in the directive contains another INCLUDE statement. For example:

```
INCLUDE     MATHFLE.ASM
```

causes the assembler to assemble the source statements of MATHFLE.ASM into the current assembly. If MATHFLE.ASM contains another INCLUDE directive, such as:

```
NAME        MATHFLE.ASM

INCLUDE     FLOAT_POINT.ASM
```

an error message is generated by the assembler.

GROUP

***Format:* name GROUP seg name [,...]**

The GROUP pseudo-op combines the specified segments (seg names) under one name, so that they all reside in one logical 64K segment. The total length of the combined segments must not exceed 64K.

The GROUP pseudo-op is another example of a directive which supports the concept of modularized programming concepts. For example, you could designate two or more code segments within a given module. Perhaps you want these code segments and code the segment defined in another module to be combined and to all reside in same segment. Figure 5-3 illustrates this. By using the GROUP statement, each specified segment is combined by the assembler into one segment.

The name field consists of an identifier used by the assembler for combining the seg name entries in the statement. In Figure 5-3, the segments are logically combined under the name: MAIN_SEG.

In this example source listing, all references are made via symbolic names; there are no absolute references in the source file.

Module 1

```
MAIN_SEG GROUP A_SEG,B_SEG
A_SEG    SEGMENT
       ASSUME    CS:MAIN_SEG
             ..
           ..
             ..
A_SEG    ENDS
B_SEG    SEGMENT
       ASSUME    CS:MAIN_SEG
             ..
             ..
             ..
B_SEG    ENDS
```

Module 2

```
MAIN_SEG GROUP C_SEG
C_SEG    SEGMENT
  ASSUME CS:MAIN_SEG
             ..
             ..
             ..
C_SEG    ENDS
```

Figure 5-3 The GROUP Pseudo-op

.RADIX

***Format:* .RADIX expression**

.RADIX allows you to specify a new default numbering base. The assembler's standard default is base 10, or decimal. Since the assembler normally uses base 10 as the default base, constants entered in binary, octal, or hexadecimal must use a suffix character to denote the numbering base of the operand.

Example:

```
MOV    AX,0FFFFH           ;'H' for hexadecimal

MOV    AL,277Q             ;'Q' for Octal

MOV    AL,00110001B        ;'B' for binary
```

By using the .RADIX directive, the default numbering base is changed and subsequent entries in that base will not require the qualifying base suffix character.

Example:

```
.RADIX     16              ;Change to hex

MOV    AX,0FFFF            ;DATA IS IN HEXADECIMAL

.RADIX      2              ;CHANGE TO BINARY

MOV    AL,00110011         ;DATA IS IN BINARY.
```

END

Format: **END [expression]**

The END directive specifies the end of your source module. If you use an expression in the operand field, then the assembler uses the expression as the starting address of the program. Do not use an expression if the module is not the main program module. Figure 5-4 illustrates the use of END in the main module and a subordinate module.

In Figure 5-4, notice that only the main module requires the END pseudo-op with a label. The label (START) defines the entry point of the program, where program execution begins after MS-DOS loads the program. Subordinate modules require the END pseudo-op only, without a label.

Conditional Pseudo-ops

Often during program development you're faced with the problem of how to assemble one portion of code if a certain condition is true and another if the

In the main module;

```
MY_CODE          SEGMENT
START:           PUSH DS
                   . .
                   . .
                   . .
MY_CODE          ENDS
                 END   START
```

In a subordinate module;

```
MORE_CODE        SEGMENT
START_MORE:      MOV   CX,AX
                 . .
                 . .
                 . .
                 . .
MORE_CODE        ENDS
                 END
```

Figure 5-4 Use of the END pseudo-op

condition is not satisfied. Conditional pseudo-ops give you this flexibility. A common misunderstanding among novice programmers is that these statements are run-time conditionals, that they apply to an assembled program when it is run on the target machine. Conditional directives do not influence the manner in which the program executes, but only what portions of code are assembled at the time of assembly.

Table 5-3 lists the conditional pseudo-ops available to the programmer using the IBM Macro Assembler. The conditional pseudo-ops test for a condition to be true or false.

You can set the condition at the beginning of the module by using the = directive. For example:

```
COMP_TYPE_A = 1

COMP_TYPE_B = 1

COMP_TYPE_C = 0
```

can be used to set conditions for the assembly of different portions of code for one of three different computer types: A, B, or C. You can then use the conditional directives to assemble the correct portions of code:

```
     IFE COMP_TYPE_A

;Assemble this code section if COMP_TYPE_A = 0
```

```
    ..

    ..

    ENDIF

    IFE COMP_TYPE_B

;Then assemble this portion of code if COMP_TYPE_B = 0

    ..

    ..

    ENDIF

    IFE COMP_TYPE_C

;Then assemble this portion of code if COMP_TYPE_C = 0

    ..

    ..

    ENDIF
```

Table 5-3
Conditional Pseudo-ops

Pseudo-op	Condition/Function
IF expression	True if expression is not 0
IFB <argument>	True if the argument is blank.
IFE expression	True if the expression is 0
IFNB <argument<	True if argument is not blank.
IF1	True if pass one.
IF2	True if pass two.
IFIDN <argument 1> <argument 2>	True if arg. 1 = arg. 2
IFDEF symbol	True if the symbol has been declared external by EXTRN pseudo-op.
IFNDEF symbol	True if symbol is not declared external via EXTRN pseudo-op, or if symbol is undefined.
IFDIF <argument 1> <argument 2>	True if string arguments are different.
ELSE	Optional clause allows alternate code to be generated when the condition is not satisfied. Only one ELSE per IF allowed.
ENDIF	Conditional block terminator. Each IF must have an ENDIF.

The previous example illustrates how one or more portions of code will be assembled if the specified expression is true. In the preceding example, an expression is true if it is equal to zero (IFE conditional used). This condition is satisfied in the example for COMP_TYPE_C only. Types A and B are equal to one; therefore, the expressions are evaluated as being false.

You can nest the conditional statements as in:

```
     IFE COMP_TYPE_A

;assemble this portion of code if true

    MOV AL,DATA_BYTE         ;Get data to send.

         IFE SERIAL_PORT1    ;Serial port1 true?

    MOV DX,SER_PORT1         ;If yes, then get port address

         ELSE                ;If it is not true then assemble
                             ;using second port address

    MOV DX,SER_PORT_2        ;Get port number

         ENDIF               ;Closes inner conditional

    ENDIF                    ;Closes outer conditional.
```

The use of the ELSE clause allows an alternate portion of code to be assembled if the preceding IF clause is not satisfied. Each nested conditional must include an ENDIF clause which closes the conditional block properly.

Listing Pseudo-ops

The listing pseudo-ops allow you to control the manner in which various listings are output by the assembler.

Table 5-4 depicts the listing pseudo-ops and their functions. For example, the .XLIST can be used to halt the listing of source and object code. To re-enable the listing, simply use the .LIST pseudo-op.

Example:

```
 .XLIST                      ;Suppress listing

 INCLUDE     DOS.EQU         ;Include the DOS equate file

 .LIST                       ;Enable the listing
```

Here the listing is suppressed so as not to list the entire DOS.EQU file. It is not necessary that every module using the DOS equates also lists them. To do so would create redundancy in your documentation, as several module listings would contain the same source statements.

Table 5-4
Listing Pseudo-ops

Pseudo-op	Function
.CREF	Enable cross reference output.
.XREF	Disable cross reference output.
.LALL	List complete macro text.
.SALL	Suppress listing of macro text, and macro object code.
.XALL	List only source lines which generate object code.
.LIST	Enable listing output.
.XLIST	Disable listing output.
%OUT text	Display the text entry during an assembly.
PAGE [length], [width]	PAGE with no arguments = go to the top of the next page. With arguments, will set page length and/or page width.
SUBTTL text	Specifies subtitle to be printed on the second line of every page of the listing.
TITLE text	Specifies the title to be printed on the first line of every page of the listing.
.LFCOND	List conditional blocks which evaluate as false.
.SFCOND	Suppress the listing of conditional blocks which evaluate false.
.TFCOND	Toggles the control switch for listing false conditionals. /X option will reverse the effect of the .TFCOND in a source file.

%OUT

Format: **%OUT text**

The %OUT pseudo-op allows a message to be sent to the display during assembly. It is most often used when assembling a large source file, and you want to monitor the progress of the assembly. By using the %OUT directive you can tell when a portion of code is being assembled, or not being assembled, as is the case when the directive is used during a conditional assembly.

Example:

```
IFE COMP_TYPE_A                           ;Is the assembly for type A?
%OUT Now assembling for TANDY 2000        ;Then display comp. type
..                                        ;etc. for other computer types
..
    ELSE
IFE   COMP_TYPE_B
%OUT Now assembling for COMPAQ
..
..
    ELSE
IFE   COMP_TYPE_C
%OUT Now assembling for the IBM PC
..
..
        ENDIF
    ENDIF
ENDIF
```

.XCREF and .CREF

Format: .XCREF
.CREF

These pseudo-ops suppress the cross-reference information (.XCREF) and enable the cross-reference listing (.CREF). If no cross reference listing has been specified, these pseudo-ops have no effect.

.LALL, .SALL and .XALL

These pseudo-ops control the manner in which macro expressions are listed. .LALL tells the assembler to list the entire macro text (including comments) during macro expansions. .SALL suppresses the listing of all text and object code produced by macro expansions, and .XALL permits only source lines that generate object code to be listed. Macros are discussed in Chapter 6.

.LFCOND, .SFCOND and .TFCOND

This set of pseudo-ops controls the listing output of false conditional blocks. .LFCOND enables the listing of conditional blocks that are evaluated by the assembler as being false. .SFCOND suppresses the listing of conditional blocks that evaluate as false, and .TFCOND toggles the current setting for the listing of false conditional blocks. Pages 5-79 through 5-82 of the IBM Macro Assembler manual detail these directives.

PAGE

Format: **PAGE [length] [width]**

The PAGE pseudo-op formats the listing file generated by the assembly. It can be used to force a top of page or to set the page's length and the number of characters printed on a line. When no operands are specified with the PAGE directive, the printer performs a form feed to the next top of page. The page number is incremented by 1, and the listing continues.

When operands are specified, they set the number of lines printed per page and the total number of characters printed per line. A special operand (+) increments the chapter number and sets the page number equal to one. The format of the page number printed at the top of each assembly listing is:

```
PAGE CHAPTER # - PAGE #
```

To increment the chapter number, use the + operand with the PAGE directive.

Examples:

```
PAGE 55,132             ;Sets page length to 55 lines
                        ;and page width to 132 chars.
                        ;per line.

PAGE 60,80              ;Sets page = 60 lines_page
                        ;and 80 chars_line

PAGE                    ;Go to the next page.

PAGE +                  ;Inc. chapter number.
```

TITLE

Format: **TITLE text**

TITLE defines the title of the listing and is listed on the first line of each page. If you do not use the NAME directive, as explained previously, the first six characters of the title text are used as the module's name. You can enter up to 60 characters in the text field. You can use only one TITLE pseudo-op per assembly.

Example:

```
TITLE     Number base conversions.
```

```
**** SUBTTL ****
```

```
Format: SUBTTL text
```

The subtitle pseudo-op specifies a subtitle, which is printed on the line immediately following the title. Unlike the TITLE directive, which is limited in use to once per assembly, the SUBTTL pseudo-op can be used as many times as necessary.

Example:

```
SUBTTL    Version 1.1.0
```

Operators

The IBM MACRO Assembler allows the use of operators within many of the source statements of a program. The operators alter the manner in which the statement is assembled. The operators can be classified as attribute, value returning, record specific, and those used to form arithmetic, logical, and relational expressions. The value-returning operators SEG, OFFSET, TYPE, SIZE, and LENGTH were discussed at the beginning of this chapter, as were the record specific operators SHIFT, MASK, and WIDTH.

Attribute Operators

Pointer, Segment, SHORT

PTR

Format: **newtype PTR exp**

The pointer operator overrides the type or distance of the label. I have already used this operator in previous programming examples to demonstrate how byte values can be referenced in data fields that have been defined with another type attribute (word, double word, etc.). For example:

```
DECIMAL_TABLE    DW     10000,1000,100,10
```

defines a table of decimal values with a TYPE attribute of word. By using a statement such as:

```
MOV    AL,BYTE PTR DECIMAL_TABLE
```

you are able to access any single byte within the table. If you were to use a statement like:

```
MOV    AL,DECIMAL_TABLE
```

an error message would be generated, since the TYPE attribute of the variable does not match the attribute being used in the source statement. Since AL is a byte-wide register, the DECIMAL_TABLE cannot be referenced without the use of the PTR operator.

The LABEL pseudo-op and the operator THIS (discussed in a moment) can also be used to allow the mixing of data types in variable access. Some examples are:

```
BYTE_DEFINE_TABLE     LABEL      BYTE

DECIMAL_TABEL   DW    10000,1000,100,10
```

or

```
BYTE_DEFINE_TABLE     EQU   THIS BYTE

DECIMAL_TABLE   DW    10000,1000,100,10
```

These would both allow the DECIMAL_TABLE to be accessed using byte-wide type instructions as in:

```
MOV    AL,BYTE_DEFINE_TABLE
```

The segment and offset attributes of DECIMAL_TABLE are assigned to BYTE_DEFINE_TABLE, while the TYPE attribute is changed from word to byte.

Similarly, the distance attribute (NEAR or FAR) of a label can be modified by using the PTR operator:

```
FAR_ENTRY EQU     FAR     PTR     LABEL_ONE

LABEL_ONE:        PUSH AX                          ;SAVE ACCUMULATOR
```

This allows either intersegment or intrasegment calls and jumps to reference the same entry point. If the label FAR_ENTRY was not defined, only intrasegment code transfers could take place, as the label LABEL_ONE is assigned a NEAR distance attribute due to the colon suffix used in its label definition.

THIS

***Format:* THIS attribute or type**

The THIS operator assigns to an operand the distance attribute (NEAR or FAR) at the current offset of the assembler's location counter. It can also be used to assign an operand the current type and segment value designated by the location counter.

Examples:

```
BYTE_VALUE        EQU  THIS BYTE          ;assigns BYTE_VALUE
                                          ;a type attribute of BYTE

MY_TABLE    DW    100 DUP(10)             ;to access bytes in table

..

PUBLIC            CODE_ENTRY              ;Declare as public

CODE_ENTRY        EQU THIS FAR            ;Allows another segment
                                          ;entry to this portion of
                                          ;code.
```

HIGH and LOW

***Format:* op-code operand1**
HIGH operand2
opcode operand1
LOW operand2

The HIGH and LOW operators allow you to isolate either the high or low bytes of a number or an address expression. This operator allows the mixing of data types. For example:

```
WORD_VAL   DW   10E0H
```

and

```
MOV   AH,HIGH   WORD_VAL
```

move the high order byte 10H into the AH register. These operators were provided to allow some degree of compatibility with the earlier Intel 8080 microprocessor.

Segment Override Operators

DS, SS, ES

When I discussed the 8088 instruction set in the last chapter, I mentioned how certain registers are automatically associated with each of the segment registers. At times this can prove to be inconvenient. What if you want to use the contents of the BP register to access data in the current data segment rather than the stack segment BP is usually associated with? Fortunately, you are able to override this association and are able to use the index, pointer, and base registers to reference data in other segments.

For example, assume BP contains 20H:

```
MOV    AX,DS:DECIMAL_TABLE[BP]
```

This moves the 32nd word entry contained in DS:DECIMAL_TABLE into AX. Other examples of segment override usage are:

```
MOV    CL,BYTE    PTR ES:[SI]     ;get byte value from
                                  ;ES:SI

MOV    AX,DS:[BP]                 ;get word value from DS:BP
```

Distance Attribute

SHORT

Format: **JMP SHORT target**

The SHORT attribute instructs the assembler to generate a 2-byte jump instruction rather than the normal 3-byte jump instruction. The program transfer address must be between +127 and −128 bytes from the jump instruction. This implies that the label attribute of the target operand is of the NEAR type.

Example:

```
    JMP   SHORT    THAT_LABEL      ;MUST BE  < (+) 127 BYTES

    ..                             ;FORWARD

THAT_LABEL:     MOV CX,NEW_COUNT   ;
```

Expressions: Relational, Logical, Arithmetic

Relational Operators

EQ, NE, LT, LE, GT, GE

The relational operators test two operands and return a zero result if the comparison is false and a word of all ones (0FFFFH) if the comparison is true. They are usually combined with a bit mask, giving you a means to manipulate an instruction (Table 5-5).

Table 5-5
Relational Operators

Format	Function
argument-1 EQ argument-2	True if argument-1 = argument-2 (equals)
argument-1 NE argument-2	True if argument-1 < > argument-2 (not equal)
argument-1 LT argument-2	True if argument-1 < argument-2 (less than)
argument-1 GT argument-2	True if argument-1 > argument-2 (greater than)
argument-1 LE argument-2	True if argument-1 < argument-2 or if argument-1 = argument-2

For example, if the label VALUE is found to be equal to 10:

```
OR  AX,((VALUE EQ 10) AND 7)  OR  ((VALUE NE 10) AND 8)
```

it reduces to:

```
OR    AX,7
```

and, if VALUE is not equal to 10, the statement is assembled as:

```
OR    AX,8
```

Logical Operators

AND, OR, XOR, NOT

I purposely used the 8088 OR instruction and a logical OR operator in the example above. The second OR is the logical operator and is evaluated at assembly time, whereas the first OR is the instruction generated by the assembler and is executed by the 8088 during program execution.

Since the assembler contains the AND, OR, XOR, and NOT logical operators, it is important that you do not confuse them with the 8088 instructions of the same name. Although they perform the same logical functions, the operators are evaluated by the assembler, and the instructions are evaluated and executed by the 8088. This is a subtle but noteworthy distinction.

Arithmetic Operators

MOD, SHR, SHL, +, −, *, /

The arithmetic operators perform assembly time calculations on the specified operands. MOD divides one operand by another and returns the result. For example:

```
VALUE1    EQU 10

VALUE2    EQU 3

   MOV    AX,VALUE1 MOD   VALUE2

evaluates to;

   MOV    AX,1
```

The SHR and SHL operators shift values either right or left a specified number of bit positions and operate in a manner similar to the shift instructions in the 8088's repertoire. However, the logical instructions, which have the same names, should not be confused with the SHR and SHL operators. Remember that the operators are used by the assembler for calculating a value to be used in an instruction or pseudo-op.

The remaining arithmetic operators perform the basic math functions of addition, subtraction, division, and multiplication. Some examples of their usage are:

```
MOV    AX,16*20                    ;Moves 420 into AX

MOV    AX,VALUE1/VALUE2            ;Moves the quotient of the division
                                   ;into AX

MOV    CX,TABLE_END-TABLE_START    ;Number of bytes in the table
```

The instruction:

```
MOV   CX,SIZE TABLE
```

also loads the CX register with the total number of bytes in the table.

The order in which operators are evaluated in an expression is discussed in the IBM Macro Assembler manual on pages 4-20 and 4-21.

In the next chapter I will discuss the Macro pseudo-ops and their usage. Included in the chapter is a complete Macro library for MS-DOS 2.0. The library is also included on diskette and should enable you to write programs for the IBM PC and compatibles in a more efficient and structured manner.

Chaper 5 Review

1. ASSUME instructs the assembler which ____________ registers to associate with each segment in the program.
2. A procedure can either be of the ____________ type or the ____________ type.
3. Match the pseudo-ops which define data for each type listed below.

   A. Ten Bytes
   B. Byte
   C. Double Word
   D. Quad Word
   E. Word

   F. DB
   G. DT
   I. DQ
   J. DW
   K. DD

4. Which pseudo-op allows redefinition of a symbolic name with a value?

   A. EQU

   B. =

5. To increment the chapter number of a listing use the ____________ directive.

6

Macros and MS-DOS

Two very important concepts are discussed in this chapter, macros and the IBM operating system, PC-DOS/MS-DOS. I've taken the liberty of combining the two subjects in one chapter for a good reason. Included in the chapter is a discussion of a program containing macros for all of the MS-DOS and many of the BIOS calls used by the IBM. By using these predefined macros, your programming time and productivity will be increased. Because the macro definitions invoke MS-DOS function calls, this chapter seemed to be a logical place to discuss both.

By the end of this chapter, you will know what macros are, how to create your own macro definitions, and how to use them. You'll also understand the manner in which MS-DOS manages the resources in your IBM PC. After reading the first part of this chapter, which explains macro definitions, you should reread the macro listing pseudo-ops discussed in Chapter 5.

Macro Definitions

Macros allow you to assign a series of often used source statements to a macro name, and then reference the entire sequence by specifying the macro name in the op-code field of a source statement. As an example, let's say you need to repeatedly use the following instruction sequence:

```
PUSH AX

PUSH BX

PUSH CX

PUSH DX

PUSHF
```

You would normally code the entire sequence every time it is need in a program. By assigning the sequence to a macro definition, the instructions are defined once, and the macro name is used to generate the instruction sequence. You define a macro in the following manner:

```
SAVE_ALL_REGS MACRO            ;Macro definition header

     PUSH AX                   ;Body of the macro

     PUSH BX                   ;    |

     PUSH CX                   ;    |

     PUSH DX                   ;    |

     PUSHF                     ;   \ /

     ENDM                      ;ENDM is the macro terminator
```

The macro is assigned the name SAVE_ALL_REGS by the macro pseudo-op. The ENDM pseudo-op terminates the macro. Anytime you need to save all the registers in your program, use the macro name SAVE_ALL_REGS in the op-code field of an assembly language statement. Figure 6-1 demonstrates how this macro and a macro definition that restores all the registers can be used in a program.

Dummy Arguments

Optional dummy arguments can also be specified in the macro definition, as depicted in Figure 6-2.

Notice in Figure 6-2 that the assembler places a plus sign to the left of any source statements that appear as a result of the expansion.

This particular macro defines a routine that places a string of characters typed from the keyboard in a predefined buffer area. The dummy argument KBDBUFFER is replaced by the name of the keyboard buffer used in the program. If your program defines a buffer area in the data segment and you name this buffer MY_KEYS, you would invoke the macro by specifying the macro name and the buffer name as follows:

```
@KBLINE       MY_KEYS
```

```
SAVE_ALL_REGS    MACRO        ;Macro header
        PUSH AX               ;Body of the macro
        PUSH BX
        PUSH CX
        PUSH DX
        PUSH SI
        PUSH DI
        PUSH DS
        PUSH ES
SAVE_ALL_REGS    ENDM         ;Macro terminator

 GET_ALL_REGS    MACRO
        POP ES
        POP DS
        POP DI
        POP SI
        POP DX
        POP CX
        POP BX
        POP AX
 GET_ALL_REGS    ENDM
```

Figure 6-1 Example Macro Usage

```
  Define the macro:
@KBDLINE  MACRO  KBDBUFFER
     MOV  AH,F_KBDLINE          ;MSDOS function 0AH
     LEA  DX,KBDBUFFER          ;Address of user buffer
     INT  MSDOS                 ;Msdos type 21H interrupt
   ENDM                         ;End of macro

  Invoke the macro in the source program:

@KBDLINE  MY_KEYS

  At assembly time the assembler expands the macro as:

@KBDLINE  MY_KEYS
 +   MOV  AH,F_KBDLINE          ;MSDOS function 0AH
 +   LEA  DX,MY_KEYS            ;Address of user buffer
 +   INT  MSDOS                 ;Msdos type 21H interrupt
```

Figure 6-2 Macros which contain dummy arguments

The assembler expands the macro definition above by substituting the statements which comprise the macro. The statements are inserted in the assembly at the point where the macro definition is encountered. The real buffer name replaces the dummy argument (KBDBUFFER) in the expansion. You can use more than one dummy argument in a macro definition by separating each argument by a comma as shown in Figure 6-3.

```
    Define the macro:

@SET_INT_VECTOR    MACRO    TYPE,SEG_VECTOR,OFFSET_VECTOR
        PUSH   DS                        ;Save old data segment
        MOV    AX,SEG_VECTOR             ;Segment of new interrupt vector
        MOV    DS,AX                     ;Place in DS
        MOV    DX,OFFSET_VECTOR          ;Offset of the service routine
        MOV    AL,TYPE                   ;The MSDOS type number of the interrupt
                                         ;vector to change
        MOV    AH,F_SET_INT_VECTOR       ;MSDOS function 25H to set new
                                         ;interrupt vector
        INT    MSDOS                     ;MSDOS interrupt 21H
        POP    DS                        ;Restore old data segment
        ENDM

    Invoke the macro:
@SET_INT_VECTOR    MACRO    0CH,MY_CODE,RS_INT_1
```

Figure 6-3 Macros with multiple arguments

Figure 6-3 shows how multiple dummy arguments are passed to a macro. Simply separate each with a comma. You may optionally not specify one or more of the parameters when you invoke the macro; in which case, the value for that parameter is set to zero (null value). In the example shown, a macro is defined which, when invoked, changes the interrupt vector for the specified interrupt type. The type number specified is 0CH, which happens to be the routine for the serial I/O board of the IBM PC.

Macros extend the capabilities of the assembler by allowing you to define your own op-codes for often used instruction sequences. Don't confuse macro definitions with procedures. Macros generate in-line code (by macro expansion) every time the macro is invoked. Procedures define a set of instructions which, when assembled, reside in only one portion of memory and are called when needed.

Macros are used when the instructions are few in number, yet often used, or when calling a procedure would cause an unacceptable delay in processing speed, the processor has to push and pop the instruction pointer and perhaps the code segment register from the stack.

Since one macro statement can be used in place of several instructions, source programs that use macros are more readable and shorter in length. The object module is usually longer, due to the in-line code produced by the macro expansions.

Macro definitions are usually defined in a separate source file, thereby creating a library of definitions. The file is included in the assembly via the INCLUDE directive, as shown in Figure 6-4.

```
Page 60,132
TITLE ANY_PROGRAM
SUBTTL USE_OF_MACRO_INCLUDE

.XLIST
INCLUDE   B:DOSEQU.EQU
.LIST

IF1
INCLUDE B:MACFLE.MAC
ENDIF
```

Figure 6-4

Notice that the INCLUDE directive, which specifies the macro library, appears within a conditional assembly block. The IF1 pseudo-op includes the macro library on the first pass of the assembler but not on the second pass. The file must be included on the first pass, so that the assembler can resolve any macro definitions referenced in the program. However, the IF1 pseudo-op prevents the macro library from being read on the second assembler pass, which generates any listings that have been specified. Therefore, the macro library does not appear in the listing file.

You might wonder why this is such a big deal. The reason is quite simple. Assume that you have created a program comprised of multiple modules. Each module utilizes macro definitions as defined in the macro library MACFLE.MAC (Listing 6-1 in Appendix D). As each module is assembled separately, the library file would be included in every module's listing! Needless to say, you need to list the macro library only once (as a source file), not every time an assembly listing is generated.

Local Labels

Labels used in a macro definition must be declared local to the macro, by using the LOCAL pseudo-op. LOCAL informs the assembler that the label is part of a macro definition and, as such, the label should be changed each time the macro is invoked. The reason the label must be altered is that the assembler requires that a label be defined only once in a source program.

Consider the following source statements:

```
@INPORT       MACRO
HERE:   IN AL,DX             ;INPUT CHAR. FROM A PORT
    CMP AL,1BH               ;ESC CHARACTER?
    JNE HERE                 ;NO? THEN LOOP
    ENDM
    @INPORT                  ;Invoke a macro for port input
    @INPORT                  ;Do it again.
```

When expanded, the macros produce the following code:

```
HERE:   IN AL,DX             ;INPUT CHAR. FORM A PORT
    CMP AL,1BH               ;ESC CHARACTER?
    JNE HERE                 ;NO?, THEN LOOP
HERE:   IN AL,DX             ;INPUT CHAR. FROM A PORT
    CMP AL,1BH               ;ESC CHARACTER?
    JNE HERE                 ;NO? THEN LOOP
```

What do you think the assembler will do? Obviously, the assembler will not be able to distinguish which label HERE the program should branch to. In this case, the assembler generates an error message to inform you that it has encountered a symbol with multiple definitions—HERE.

Since macros are usually used more than once in a program, any labels appearing within the body of the macro must be declared local to it. Failure to declare them local causes the assembler to generate an error message. LOCAL must be used immediately after the macro statement. It must be used before any other statements in the macro, including comments. Figure 6-5 illustrates the use of the LOCAL pseudo-op.

```
@SET_C_CHECK    MACRO   SWITCH1,SWITCH2
        LOCAL   FETCH
        MOV     AL,SWITCH1              ;Set or fetch control C kbd
        OR      AL                      ;Checking. Z means fetch state
        JZ      FETCH
        MOV     DL,SWITCH2              ;Set the state
FETCH:
        MOV     AH,F_SET_C_CHECK        ;MSDOS function 33H
        INT     MSDOS                   ;Interrupt 21H
        ENDM
```

Figure 6-5 Use of the LOCAL pseudo-op

When expanded, the assembler generates a unique label in place of the label specified as LOCAL (FETCH, in this example). This prevents a multiply-defined symbol error message from being generated on repeated macro expansions.

Other Macro Pseudo-ops

Other macro pseudo-ops facilitate the definition of macros. Table 6-1 lists the macro pseudo-ops available with the IBM Macro Assembler. Additionally, any of the conditional pseudo-ops discussed in Chapter 5 may also be used in the body of a macro definition. I'll discuss some macros that use conditional pseudo-ops later in this chapter.

Table 6-1
Macro pseudo-op definitions

Pseudo-op	Function
name MACRO dummylist	Macro header. Optional arguments may be specified in the dummylist. A macro name must always be specified.
ENDM	Macro terminator. Ends macro definition.
EXITM	When encountered, the macro expansion terminates immediately.
LOCAL dummylist	When encountered, the assembler will create a unique name for each occurrence of any argument listed as a dummy parameter.
REPT expression	When encountered by the assembler, any statements between REPT and ENDM are repeated by the number of times specified in the expression field.
PURGE macro-name	Deletes the specified macro from the assembly, allowing the memory the definition occupied to be reused.
IRP dummy, <argument-list>	Specifies a number of elements (argument-list), which are to be substituted for each occurrence of the dummy parameter within the block. (See text for a more complete definition).
IRPC dummy,string	Specifies a number of characters which are to be substituted for each occurrence of the dummy parameter within the block.
	Code produced by macro expansions.
	Object code in the expansion.

continued

Table 6-1
Macro pseudo-op definitions

Macro Operators	Function
text&text	Concatenates text or symbols in the macro expansion.
;;text	Inhibits the listing of a comment following the double semi-colon.
!character	The character following the exclamation point is entered literally in the expansion.
%expression	Converts the expression to a number using the current radix.

PURGE

Format: **PURGE macroname[,....]**

The PURGE pseudo-op deletes the specified macro definitions from the assembly. The assembler deallocates the memory space it has reserved for the macro definition. Once a macro has been purged, the memory space it occupied can be used by the assembler for other storage requirements.

If you purge a macro from the assembly and then reuse the macro definition, the assembler generates an error message. You can use PURGE to delete macros that are never used in a program module, or you can use the directive to delete a macro if it will not be used again in the source module.

REPT

Format: **REPT expression**

The REPT pseudo-op causes the statements appearing between REPT and ENDM to be repeated the specified number of times.

Example:

```
TABLE_GEN MACRO    INIT1,WORDS        ;Macro name and parameters

NUMBER = INIT1                        ;Set initial value

   REPT WORDS                         ;Parameter specifies # of repeats

SQUARE = NUMBER*NUMBER                ;Generate the square

NUMBER = NUMBER+1                     ;Increment to next number to square.
```

```
DW    SQUARE                       ;Define word in memory

ENDM                               ;End repeat block.

ENDM                               ;End Macro
```

This macro produces a table of squares. The macro is invoked by specifying the macro name along with the parameters INIT1 and WORDS. INIT1 is the initial value for the first number to square, and WORDS is the total number of squares to generate (also the number of word entries in the table). The macro is invoked as follows:

```
TABLE_GEN 1,10             ;Generate a 10-entry table of squares
                           ;beginning with 1.
```

The table will contain the following values: 1, 4, 9, 16, 25, 36, 49, 64, 81, and 100.

You can also use the REPT and ENDM pseudo-ops outside of a macro definition. This provides an easy means of duplicating source statements.

IRP

Format: **IRP dummy,**
<argument list>

The IRP pseudo-op causes each successive entry in the argument list to be substituted for the dummy parameter. Let's say you want to define the table of squares, as was done in the previous example, but this time using the IRP pseudo-op. The table could be generated as follows:

```
IRP   SQ_VAL,  <1,4,9,16,25,36,49,64,81,100>

DW    SQ_VAL

ENDM
```

Each value in the argument list is substituted for the symbol SQ_VAL, creating a 10-entry table of squares.

IRPC

Format: **IRPC dummy,string**

This pseudo-op lets you define ASCII character strings in memory. It is similar to the other repeat pseudo-ops, in that it causes a repetition of the block of statements appearing between the IRPC and ENDM directives. The pseudo-op can be used in a macro definition or on its own (as can the other repetition pseudo-ops), as follows:

```
IRPC    MESSAGE,Help!       ;Character string = 'Help!'

DB      MESSAGE             ;One byte for each byte in the

ENDM                        ;message.
```

The sequence above causes the following ASCII codes in the word Help! to be stored in memory as 48H, 65H, 6CH, 70H, and 21H.

EXITM

Format: EXITM

The EXITM pseudo-op causes the immediate termination of a macro expansion by the assembler. It is used after a conditional pseudo-op within the macro, which causes the assembler to encounter the EXITM pseudo-op and terminate the expansion. Figure 6-6 illustrates the use of the EXITM pseudo-op.

```
    @MY_MACRO MACRO
          IFE TRUE
          XCHG AX, BX          ;Swap AX and BX if condition is true
          EXITM                ;Then exit
          ENDIF
          XCHG AX,CX           ;Else if the condition is false,
                               ;swap AX and CX
          ENDM

To force the condition in a source program:

    TRUE = 0                   ;Set true to 0

    @MY_MACRO

+   XCHG AX,BX                 ;Swap AX and BX if condition is true
```

Figure 6-6 Use of the EXITM pseudo-op

In the example shown above, EXITM is used within a conditional block. If the label of TRUE is evaluated to be zero by the assembler, the portion of code residing in the conditional block is assembled (XCHG AS,BX). Since we set TRUE equal to zero prior to invoking the macro, the conditional block, which includes the EXITM pseudo-op, is assembled. When the assembler encounters the EXITM pseudo-op, the macro expansion is terminated as if the ENDM directive had been encountered.

Macro Operators

Several operators enable you to construct complex macro definitions. Just as the operators used in source statements modify the instructions generated by the assembler, the operators used in a macro definition affect the manner in which the assembler expands the macro.

&

Format: **& expression**

The ampersand (&) operator concatenates text or symbols found in the macro definition. It can be used to dynamically define instructions, registers, or even labels within a macro.

```
CLS   MACRO      REG,COND,LINES

CLS&REG:  MOV A&REG,"&COND"

   MOV BX,LINES

   ENDM
```

If the macro were to be invoked as:

```
CLS     H,Z,24    ;
```

it would be expanded as:

```
CLSH:        MOV AH,"Z"

     MOV BX,24
```

In this example, a register was defined by the parameter REG, the ASCII character Z was defined by the parameter of COND, and the value to be moved into the BX register was defined by the parameter LINES.

Another operator suppresses the expansion of a comment. When the assembler encounters a double semicolon (;;), the comment that follows is not expanded during assembly.

Example:

```
;;This line will not be part of the expansion.
```

%

Format: %EXpression

The percent operator (%) converts the specified expression to a number. The number returned will be in the current radix. Use of the percent operator is limited to macro arguments, as illustrated below:

```
A_MACRO    MACRO    PARAMETER_1        ;Define the macro.

     DB    PARAMETER_1                 ;Define byte storage.

     ENDM                              ;End of the macro.
```

Once the macro is defined, invoke it and use the % operator to specify the parameter, as follows:

```
X = 10                  ;NUMBER OF TIMES TO REPEAT MACRO

VALUE = 0               ;SYMBOL DEFINED FOR PARAMETER

    REPT X              ;REPEAT THE NEXT STATEMENTS 'X' TIMES

VALUE = VALUE+1         ;INCREMENT VALUE

    A_MACRO %VALUE      ;INVOKE MACRO, CONVERT VALUE TO

                        ;A NUMBER IN CURRENT RADIX

    ENDM                ;END OF REPEAT
```

The above section of code invokes the macro definition A_MACRO ten times. VALUE is substituted for the dummy argument PARAMETER_1 each time A_MACRO is invoked. Since the label VALUE is assigned a numeric value that is incremented prior to each macro invocation, the sequence of instructions produces a 10-byte table of 01H through 0AH.

MS-DOS/PC-DOS and Macros

The operating system (OS) of the IBM PC is an advanced and powerful operating system. The combination of the PC's hardware design and its operating system opened up a new world to personal computer users, programmers, and third-party hardware suppliers. The operating system was originally developed by the Seattle Computer Company and sold to Microsoft, which continually improves upon the OS, with the latest version (at this writing) being MS-DOS version 2.10.

Operating Systems

An operating system is the software that manages the resources of the computer (hardware). It is responsible for the management of I/O devices, including the video display, disk drives, and other functions, such as memory allocation.

The system is also responsible for the user interface, the manner in which the user of the system receives and requests services from the OS. Utility programs that perform specific device functions are considered to be an adjunct of the operating system. These programs usually perform such functions as formatting a diskette, copying one file to another, and so on. MS-DOS contains similar utilities and a command-line user interface, in which you type a command line from the keyboard that is actually a request for a system resource or service.

Aside from the user interface level, which allows you to converse with the operating system, MS-DOS has standardized the method of access to the functional calls within the operating system. These routines provide you with easy access to most of the major operating system functions necessary to write application programs. If you write your programs in such a manner that they interface to the hardware of the IBM (disk drives, etc.) through MS-DOS function calls, you can be assured that the programs will run on other MS-DOS computers with little if any modification.

This is the major advantage inherent to MS-DOS, device independence. Device independence means that you do not have to concern yourself with the physical aspects of the hardware for a given machine. It does not matter to you if the printer port for the computer is at port 03C0H or at 0030H. You need only know that by using the appropriate MS-DOS function call, the character in the register will be sent to the printer if one is attached to the system.

MS-DOS accepts user requested function calls and links those requests to lower level functions that are machine specific. The function calls requested are linked to BIOS, the Basic Input/Output System of the IBM PC. BIOS is responsible for the actual hardware interface of a given machine.

Although this description is greatly simplified as to how MS-DOS and BIOS interact, it illustrates the fact that the BIOS implementation may be slightly different from one compatible computer to the next. However, MS-DOS has been standardized in such a manner that you are somewhat isolated from the hardware differences.

The following discussion details the functional calls found in MS-DOS and their usage. The information describing these calls in IBM's *DOS Technical Reference Manual* is minimal at best and is not well suited to novice programmers.

All of the functional calls in MS-DOS lend themselves nicely to being defined as macros. Listings 6-1 and 6-2 (see Appendix D) define the MS-DOS macro calls used to invoke MS-DOS function calls. Before looking at the programs, let's look at the mechanism through which MS-DOS manages the IBM PC's resources.

Function Calls

When I discussed the 8088's instruction set in Chapter 4, I described the INT instruction as a shorthand method of calling a procedure. When the 8088 encounters the INT instruction, the current contents of the IP and CS registers are saved on the stack, and the registers are then loaded with new values that send program execution to the appropriate interrupt service routine.

The IBM operating system, PC/MS-DOS, uses interrupt types 20H - 27H. MS-DOS uses interrupt type 21H almost exclusively for executing function calls. MS-DOS expects that certain registers contain certain values, or addresses, when the type 21H interrupt routine is encountered. Specifically, AH is used as the messenger to inform the operating system of the function being requested.

The function specified might be to read a key from the keyboard into a buffer, display a character, or to write a sector of information to disk. MS-DOS interprets the value in AH as a function code and executes a specific function based upon this value.

Other registers and, for some function calls, memory areas must be initialized prior to using the function calls. For example, most disk functions require that a Disk Transfer Area (DTA) or a file control block be established in memory and that the DX register point (contain the base address) to these areas. Other registers may be required for other specific purposes. Each function call uses the registers of the 8088 in a slightly different manner and for slightly different purposes.

The minimal format for executing an MS-DOS system call is:

```
MOV    AH,FUNCTION_CODE         ;Get function code

INT    MSDOS                    ;EXECUTE DOS INTERRUPT TYPE 21H
```

where FUNCTION_CODE is the MS-DOS function request number, (read from keyboard, display a character, etc.) and the label MSDOS is equated to 21H, the MS-DOS interrupt type number for function requests.

If the instruction sequence depicted in the preceding example is used often within a program, it would save you a considerable amount of time if only one source statement had to be typed, as opposed to the two shown above. By defining the instruction sequence as a macro name, only the macro name needs to be entered into the source file where the instruction is required, as follows:

```
@WAITKEY   MACRO
     MOV   AH,F_WAITKEY       ;FUNCTION REQUEST
     INT   MSDOS              ;DOS CALL
     ENDM
```

Now the instruction sequence is assigned to the macro name @WAITKEY. The instruction sequence is generated (or expanded) in the assembled version of the program wherever the assembler encounters @WAITKEY.

In the example above, the symbol F_WAITKEY is used in place of the absolute value for the function. The label MSDOS is similarly defined elswhere in the file to have the value 21H; the value for the MS-DOS function call interrupt type. I have defined all such MS-DOS calls as macros in MACFLE.MAC and included them in Listing 6-1 in Appendix D.

DOSEQU.EQU, the other file of interest, is shown in Listing 6-2 in Appendix D. This file is the DOS equate file, which contains all of the equates used for the function code numbers under MS-DOS 2.10. If you use the files DOSEQU.EQU and MACFLE.MAC in your program modules, you must use the INCLUDE pseudo-op to read them into your assembly as follows:

```
.XLIST
INCLUDE    DOSEQU.EQU
.LIST
IF1
INCLUDE    MACFLE.FLE         ;Include macro file on pass 1 only.
ENDIF                         ;End of conditional block.
```

The DOS equate file must be read on both passes of the assembler or a phase error will result. Phase errors occur when the assembler finds a different value for a label, variable, or procedure on the second pass than it found during the first pass.

.XLIST and .LIST inhibit the listing of the equate file for the same reason IF1 and ENDIF are used to inhibit the macro file from being included in the listing. The difference between using the .XLIST/.LIST and IF1/ENDIF directives is that the assembler reads the DOSEQU.EQU file on both passes but reads the MACFLE.MAC file only during the first assembler pass. In either case, it is not necessary to include these files in the listing file, unless you happen to own your own paper mill and are not concerned with wasting paper when you print out the listing file.

You'll notice in Listing 6-1 that the labels BIOS and MSDOS are equated at the beginning of the macro file. I don't believe that Microsoft will ever change the interrupt type number for the function calls, but in keeping with earlier discussions of defining labels, a label is assigned the numeric value for the BIOS interrupt type and the most often used MS-DOS interrupt type. Should they change, simply change the interrupt type number assigned to the label to the new value. You won't have to edit any of the macros if the interrupt type is redefined.

Each macro in Listing 6-1 is commented as to the MS-DOS function invoked and any other special considerations you should be aware of. Look at the macro that defines function code 01H in Listing 6-1. The function waits for a key to be pressed at the keyboard and moves the character typed at the keyboard into the AL register. The character is also echoed to the display.

Other than preparing the AH register with the function code 01H, prior to executing the INT 21H instruction, no other special preparations are necessary. After executing the function, the user program would then process the character returned in AL.

Minimal Programming Considerations

Many of the MS-DOS function calls expect dedicated areas to be established in memory prior to issuing a function call. Within these dedicated areas there may be several bytes or words that you must initialize to certain values prior to invoking the MS-DOS function.

When using the character oriented keyboard functions, 01H, 06H, 07H, and 08H, you don't need to have a special RAM area established, as the character is returned in AL. However, there are two methods available to obtain a line of keyboard input. You can either repeatedly use one of the single character function calls and build the string in a buffer area or use the MS-DOS function call 0AH, which waits for a line of input to be entered from the keyboard.

If the latter function is chosen, a buffer area must be defined in the data segment as follows (Note: You can use any valid names for the labels shown, but the reserved memory must remain the same):

```
MAX_CHARS:     DB    (?)     ;Maximum characters allowed.

CHARS_TYPED    DB    (?)     ;Number of characters typed.

KBDBUFF        DB    32      ;The actual buffer must be as
                             ;large as the maximum chars
                             ;specified in MAX_CHARS.
                             ;It can be larger or smaller
                             ;than the value used in this
                             ;example.
```

The buffer area consists of a defined byte which communicates to the function 0AH, the maximum number of bytes you expect to be entered from the keyboard. If you exceed the specified number of characters when entering the input line, the function produces a tone at the speaker (beep) and won't accept any further input except for a carriage return (which must terminate the input line).

On entering the function, DS:DX must point to the beginning of the buffer area MAX_CHARS. You can use LEA DX,MAX_CHARS to load DX with the address pointer before invoking the function. The maximum number of characters the function is to return must be specified in MAX_CHARS prior to calling the function. The maximum count must include the carriage return as the terminator to the string. If the anticipated input is four characters as in GARY, then set MAX_CHARS to 5. Four characters and the carriage return will then be accepted by the function.

On return, the number of characters actually typed from the keyboard will be in the memory location CHARS_TYPED. You can use this information as a count factor to move the data from the keyboard buffer to another buffer if you like (See the MOVS instruction).

Listing 6-3 in Appendix D contains a short program that accepts a line of input from the keyboard and displays what was typed. Notice the creation of the necessary keyboard buffer area KBD_BUFFER at the beginning of the data segment. Notice also that the $ terminator is defined in memory at the label DELIMITER. I'll discuss why I used the delimiter in a moment when the display routines are discussed. The program uses macro definitions found in the macro file, MACFLE.MAC, and the function codes are part of the equate file, DOSEQU.EQU.

Example Program

The example program KEYDSP in Listing 6-3 uses the following macro calls:

```
@CLS         -> Clears the display.

@CHARDSP     -> Displays the character in DL

@VDLINE      -> Display a line of text pointed to by
                DS:DX, and terminated by $.

@CON_INPUT2  -> Wait for a character to be typed at the keyboard, and
                return it in AL.

@KBDLINE     -> Get a line of input from the keyboard.

@CRLF        -> New line function, generates a carriage
                return, linefeed, places the video cursor
                on the 1st column, next line.

@WAITKEY     -> Waits for a key to be entered from the keyboard
                and echoes the character to the display.
```

The program prompts you for keyboard input, accepts what you type from the keyboard, and displays it. The program then asks you to press any key to continue and repeats the entire program. To terminate the program, type Control-C from the keyboard, and the program ends, returning you to DOS.

Review the macro definitions in the MACFLE.MAC listing for the keyboard functions 01H, 06H, 07H, 08H, 0AH, 0BH, 0CH. All of these functions can be used to retrieve a character from the keyboard. The difference between the functions is that some automatically echo the keyboard character to the display while others do not. Some functions (such as 01H) will not return until a character is entered from the keyboard. Other functions recognize the Control-C character while others do not check for the control character. Each function is summarized in the MACFLE.MAC listing.

The Stack Segment

The stack segment is established as 100 words (200 bytes), which is more than enough for this program. If the program were more complex, the stack may have to be allocated more memory.

The Data Segment

The data segment contains all the necessary buffers and messages used in the program. The segment contains the keyboard buffer area already discussed and all of the messages used to converse with the user. The delimiter $, is required by the MS-DOS function call 09H, which displays a line of ASCII text pointed to by DX. The delimiter terminates the text to be displayed and, when encountered during the line display function, terminates the function and returns control to your program. The terminator is also required for the MS-DOS function 09H.

The Code Segment

The code segment for the program in Listing 6-3 contains instructions to make the program perform the desired task; it accepts and displays input from the keyboard. The program logic is shown in Figure 6-7.

The source program in Listing 6-3 is comprised of mainly macro statements. It is comparable to viewing a program which was written in a high level language, in that the macro names describe a function and not necessarily the instructions which comprise the function itself.

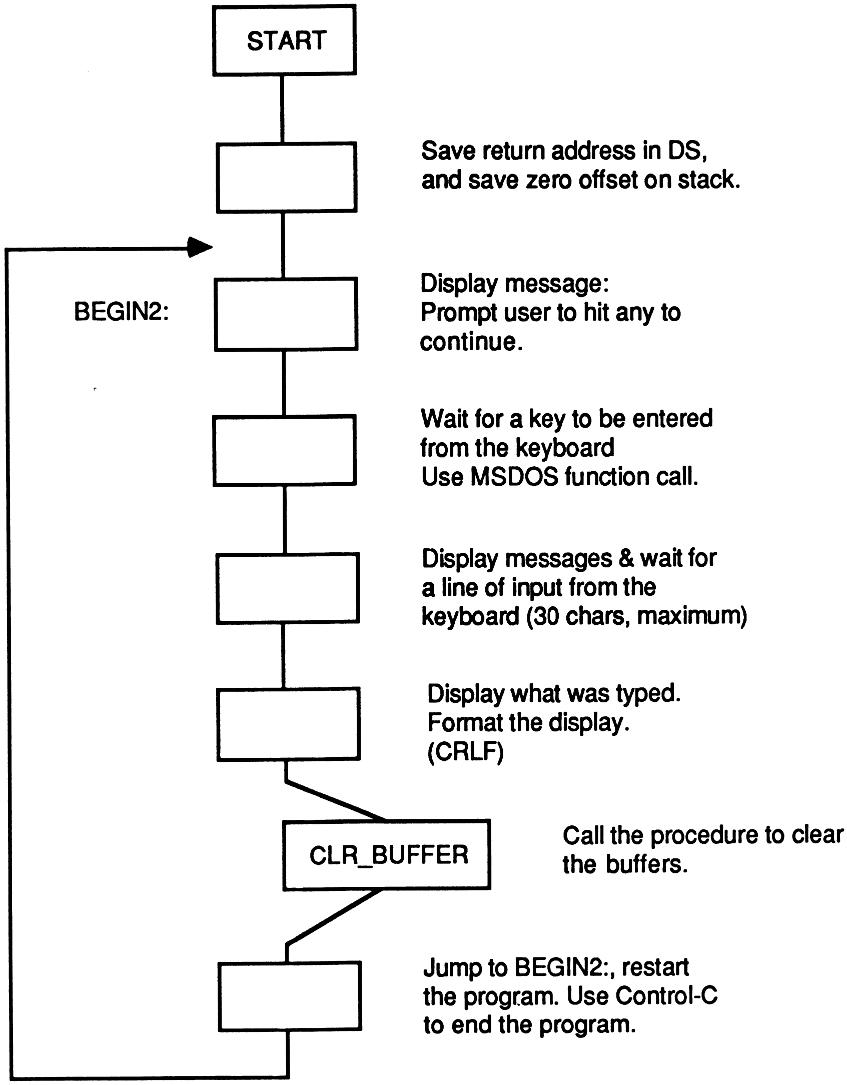

START
Save return address in DS, and save zero offset on stack.
BEGIN2:
Display message: Prompt user to hit any to continue.
Wait for a key to be entered from the keyboard Use MSDOS function call.
Display messages & wait for a line of input from the keyboard (30 chars, maximum)
Display what was typed. Format the display. (CRLF)
CLR_BUFFER
Call the procedure to clear the buffers.
Jump to BEGIN2:, restart the program. Use Control-C to end the program.

Figure 6-7 Flow Chart for KEYDSP.ASM

If you wanted to, you could define a macro language that could appear as if it were a high level language. The only disadvantage is that the instructions comprising a macro definition are expanded as in-line code every time a macro is encountered. This is not a very efficient use of memory and is not a recommended programming habit.

When repetitive instruction sequences produce more than a few bytes of code, it is better to code the sequence as a procedure and not define the sequence as a macro definition. This results in a reduction of executable code produced by the assembler.

The instructions that comprise the body of each macro are present in the assembled listing of the program. They immediately follow the macro name. The assembler places a + in column 31 of the listing, indicating that the current line was generated due to a macro expansion.

Rather than explain each of the DOS functions in detail, I'll explain how to use the functions in programming examples. The macro definitions have many comments and require little explanation here. You should examine the macro definitions in detail before experimenting with their usage. I will explain dedicated RAM buffers or areas that must be established prior to using the functions. The explanations appear in the programming examples as they are presented.

You are now ready to earn your wings as an Assembly Language programmer. In Chapter 7 I'll show you a complete programming example from start to finish, from conceptual design to actual implementation. So "Kick the tire and light the fire," let's GO!

Chapter 6 Review

1. Macro definitions assign a __________ of often used __________ to a macro name.
2. A macro is composed of a __________, a __________, and a __________.
3. ENDM __________ the macro definition.
4. To invoke a macro, use the __________ __________.
5. The LOCAL pseudo-op assigns a __________ name to a label each time the macro is invoked.
6. The minimal format for executing a MS-DOS function call is: (Write in the form below)

   __________ __________,__________

   __________ __________
7. Why do MS-DOS function calls lend themselves to macro definitions?

7

Number Conversions

A Programming Example

NUMBERSY.ASM, the program discussed in this chapter is like the others I have presented, in that the program is capable of standing on its own. It is completely interactive with the user and illustrates many of the principles that I have discussed so far. (See Listing 7-1 in Appendix D.)

For those who are still trying to comprehend how decimal, binary, and hexadecimal relate to each other, this program should answer your questions. The program converts numbers between 0 and 65,536 from one base to another.

I'll discuss the program from its beginning premise to its ending statement. This program should serve as an example for defining, implementing, and documenting your future programs.

Purpose

This program allows you to enter a number in one of three number bases, decimal, binary, or hexadecimal, and the computer translates the number to the other two bases and displays it in each of the numbering systems. This type of program illustrates the usefulness of the computer in performing repetitive and error-prone calculations.

If you recall the discussion of numbering systems and conversions from one base to another in Chapter 1, there were many repetitive steps involved in the translation from one number base to another. Why not let the computer handle the conversion? Simply choose the number base you want to convert from (decimal, binary, or hexadecimal), enter the number, and let the computer perform the conversions to the remaining numbering systems.

The program performs the following:

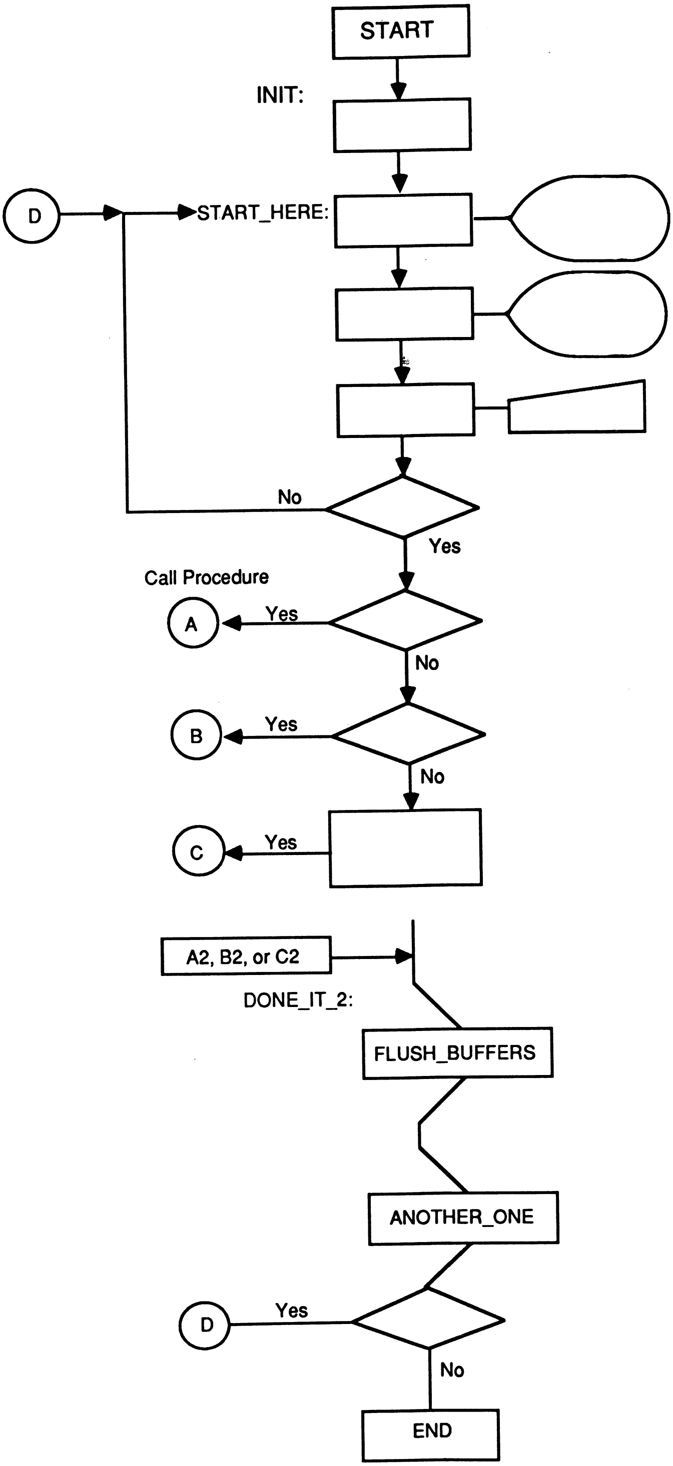
1. Accept a number in;
    - A. Decimal, and convert the number to hexadecimal and binary.
    - B. Hexadecimal, and convert the number to binary and decimal.
    - C. Binary, and convert the number to decimal and hexadecimal.
2. Display the conversions.
3. Asks you if you want to repeat the program and perform another conversion.
4. If yes, go to step 1; else exit the program and return to MS-DOS.

Conceptual Design

One of the first things you should do is sketch out the program using a flow chart or write down the programming steps. Both methods are shown in Figure 7-1. A flow chart or some other written form of the program's flow helps break the program down into its modular components. Each module performs a specific function and, as such, allows others to better understand the program and allows easier modification of the program if necessary.

The program must translate numbers entered in decimal and hexadecimal form to binary. The binary number must then be converted to the remaining number base. Similarly, the program converts numbers entered in binary to decimal and hexadecimal.

It is not enough that you are able to conceive of the algorithm, or steps required to satisfy the application, and perform the conversion. In this program, as in others you will write, the modules that support the application must be present and a part


START
INIT:
D
START_HERE:
No
Yes
Call Procedure
A
Yes
No
B
Yes
No
C
Yes
A2, B2, or C2
DONE_IT_2:
FLUSH_BUFFERS
ANOTHER_ONE
D
Yes
No
END

Save return address and offset. Establish DS and ES use.

Clear the screen

Display the prompts for the user's choice.

Get response from user.

Was the response valid? Continue if yes.

Was it decimal to binary and hexadecimal?

Was it binary to decimal and hexadecimal?

Must be hexadecimal to decimal and binary?

Run the program again.

Is the carry set? If so, then repeat the program.

Else end the program.

Figure 7-1 Flow Chart for NUMBERSY.ASM

of the program. An interactive user interface must be defined that will display messages and accept user input. Error-handling routines, or error trapping within the program modules must be specified to insure that all data entered are valid.

Other commonly used subroutines or procedures must be defined as well. Routines that clear the screen, perform bounds checking, and other functions must also be part of the program.

The macros used in this program are in the MACFLE.MAC listing (Listing 6-1 in Appendix D). The macros shorten program development time and make the source program more readable.

Segment Definitions

As illustrated in previous programming examples, there must be at least one stack segment and one code segment in every program you write. It is not necessary for every module within the program to use its own stack segment, but there must be a minimum of one stack segment defined in the program. Naturally, somewhere in one or more modules, a code segment (or segments) must be defined. Without the code segment, there would be no executable code produced.

In NUMBERSY.ASM, the stack, data, and code segments are titled MY_STACK, MY_DATA, and MY_CODE, respectively. I will discuss each of the segments as they pertain to the program.

Data Segment (MY_DATA)

The data segment reserves storage for a keyboard buffer (KBD_BUFFER) that is used to accept a line of input as entered from the keyboard. The buffer area is established to conform to the requirements of the MS-DOS function call 0AH. The function expects the maximum number of characters that will be entered to be specified in the first byte (byte 0) of the keyboard buffer. This byte is labeled in the program's data segment as MAX_CHARS.

The function accepts a line of input that is terminated by a carriage return. If the number of characters entered exceeds the number specified by the contents of MAX_CHARS, the routine rejects the character and produces a beep tone through the internal speaker. The MS-DOS function 0AH stores the actual number of characters entered in the second byte of the buffer (CHARS_TYPED). The data entered from the keyboard begins with the third byte of the buffer (KBD_BUFFER + 2) and ends with a maximum of KBD_BUFFER + MAX_CHARS. Therefore, it is necessary to define a buffer large enough to hold the maximum number of characters that will be accepted as a line of input from the keyboard.

Notice that MAX_CHARS is initialized to 32. This is done only because it is the maximum number of characters allowed in the buffer K_BUFF. When the line input function is needed in the program, MAX_CHARS should be set to the number equal to the maximum number of characters anticipated. In other words, you can dynamically change the number of characters that are to be entered from the keyboard prior to using the function. You are not limited to the value MAX_CHARS is initialized to. The only requirement is that the maximum number of characters specified does not exceed the total buffer's length.

The function number 0AH is moved to the AH register and the effective address of the keyboard buffer (KBD_BUFFER, in this example) to DX. The macro @KBDLINE accomplishs the register setup prior to invoking the function. Simply invoke the macro as follows:

```
@KBDLINE  KBD_BUFFER
```

The parameter specified in the invocation is the buffer where the keyboard input is to be stored. MAX_CHARS must be set up prior to invoking the macro.

Messages

Most programs contain a number of messages that define a portion of the user interface to the program. Messages are used to prompt you when input is required, to confirm previous input from the user, and to display error messages. In an era of "user friendly" programming, the more messages used, the easier the program will be for others to understand and make use of.

The message area for the program NUMBER_SY.ASM defines the messages necessary to obtain user input in the program. Each message is terminated by 24H ($). You may recall from previous programming examples that the terminator is required by the MS-DOS function 09H. If the terminator is not present, the function continues to display whatever characters or data it encounters until the terminator is found. MESSAGE_0 through MESSAGE_9 are the messages used in NUMBERSY.ASM.

Storage Allocation

When performing the conversions from one number base to another, certain memory locations are set aside to hold or accumulate results. When a number to be converted is entered from the keyboard, each character is ASCII encoded. For example, the string 35678, as entered from the keyboard, is in its ASCII equivalent form: 33H, 35H, 36H, 37H, 38H. Although the number can be contained in two bytes in binary form, some type of storage must be allocated for the number 35678 when it is represented in ASCII. In this example, 5 bytes of storage must be allocated for the value.

The storage locations that have been defined to hold the ASCII characters entered from the keyboard are:

```
HEXADECIMAL  - will store 4 ASCII digits entered in hexadecimal
               format (0-9, A-F) ;

BINARY_ASCII - will store 16 ASCII digits entered in binary
               format (1 or 0);

DECIMAL      - will store 5 digits entered in decimal format ( 0-9 ).
```

The word location BINARY stores the data which has been converted from ASCII to true binary form. The first operation performed by any of the three conversion routines is to convert the ASCII data entered from the keyboard to a true 16-bit binary representation. The result of this conversion is stored in BINARY.

Defining Tables

Two tables have been defined in the data segment: DECIMAL_POWERS, and HEXADECIMAL_POWERS. These tables define the powers for each digit position for the particular number base. For example the number 35678 is actually:

```
Digit 5 = 3 x 10 to the fourth power = 30000

Digit 4 = 5 x 10 to the third power  =  5000

Digit 3 = 6 x 10 to the second power =  600

Digit 2 = 7 x 10 to the first power  =   70

Digit 1 = 8 x 10 to the zero power   =    8
```

The sum of each digit's value results in the total magnitude of the number. The algorithms I chose for the conversion routines decompose decimal and hexadecimal numbers in this manner.

Equates and Other Definitions

Aside from the usual inclusion of the DOS and BIOS equate file (DOSEQU.EQU) at the beginning of the source file, the labels ONE, FOUR, FIVE, SIX, and SPACE are equated to their ASCII equivalent values. They are used in the only macro defined directly within the program.

The DISP_PROMPTS macro displays up to three messages and two numbers. It was defined to allow you an easy method of displaying a message in a predefined format. Specifically, it is used to prompt the user for the number to be converted. As each type of conversion requires a different number of digits to be entered, the macro formats the display in the following manner:

```
Message1 ---> Message2 ---> Number1 ---> Number2 ---> Message3.
```

By invoking the macro in a manner similar to:

```
@DISP_PROMPTS MESSAGE_1,MESSAGE_4,MESSAGE_5,FIVE,SPACE
```

the following prompt appears on the video display:

```
------------------------------------------------------------------
| Enter the decimal value you want to convert:                   |
| Enter 5 digits:                                                |
------------------------------------------------------------------
```

You'll find the messages MESSAGE_1, MESSAGE_4, and MESSAGE_5 defined in the data segment.

By using the macro, a prompt can be created for any of the number conversions the program is to perform. In the example just cited, the program is prompting you for a 5-digit decimal number which will be converted to binary and hexadecimal. As this macro is somewhat specialized and dedicated to this application, it is not included in the macro file MACFLE.MAC in Listing 6-1.

Code Segment MY_CODE

Initialization

The program begins with the usual setup of the data and extra segment registers. After the standard program preamble, NUMBERSY.ASM begins at the program label START_HERE.

The program begins by clearing the screen and prompting you for the type of conversion. Respond by entering the type of conversion desired. If the entry is valid, the program branches to the appropriate routine to perform the conversion. Should you enter a nonexistent choice, the program traps the entry error and asks the question again. The symbolic label INIT marks the beginning of the program; it initializes the data and extra segment registers.

The screen is cleared by moving a count of 24 into the CX register and repeatedly executing a carriage return/linefeed combination until the CX register is 0. This effectively clears the screen. The only disadvantage to clearing the screen in a manner such as this is that the cursor remains positioned at the first column of the 24th line on the CRT (bottom left corner). I'll demonstrate a better way to clear the screen and position the cursor in later chapters.

Next, the MS-DOS function to display a line of ASCII text is invoked to prompt you for the type of conversion desired. The macro @VDLINE, when expanded, invokes the MS-DOS function 09H. The parameter supplied to the macro must be an ASCII message which is defined in the data segment and terminated with 24H ($).

Program Label: RESPONSE

Once the prompts are displayed on the screen, you must enter a 1, 2, or 3. Remember that the data returned from the keyboard using the MS-DOS keyboard functions is in ASCII form and may not be in the form required by the program. The macro @WAITKEY invokes the MS-DOS function 01H; it waits for a key to be pressed, and echoes the character back to the display. The character typed will be in register AL after the function returns from MS-DOS.

After receiving the character typed at the keyboard, a linefeed and carriage return is executed, which moves the cursor to the start of the next line. By moving the cursor to the next line, the display remains neatly formatted. It is quite easy to forget these little necessities when programming. But remember if your program does not look nice to the user or it is difficult to use, you'll find yourself writing programs that can only be used by one person—you!

I always try to write programs (even ones that only I will use) as if they were commercial programs or routines I had purchased for myself. I find I am also my own worst critic. Therefore, the program must not only be aesthetically pleasing, but it must perform exactly as intended. If you write your programs in a similar manner and judge them as if you had purchased them from another party, you'll write solid programs that can be used by almost anybody.

The value entered from the keyboard is in the AX register and is temporarily saved on the stack prior to executing the carriage return/linefeed combination. The value is restored (popped) after the newline function is executed. Since the macro @LFCR destroys the contents of the AL register, the register must be saved before using the macro.

The program executes one of three possible routines, based on your entry. You could have the program simply compare the value that was entered to its ASCII counterpart to determine which routine to execute. However, I've used another and more efficient way to accomplish a branch in the program. Let's say you type a 3 from the keyboard. First, the high order bits are masked by logically ANDing the AL register with 0FH. The effect of masking the high order digits is then:

```
Value in AL   --->>> 0  0  1  1  0  0  1  1   (ASCII 3 = 33H)

AND with 0FH --->>> 0  0  0  0  1  1  1  1
                   ------------------------
Result        -->>>- 0  0  0  0  0  0  1  1   (Binary 3)
```

What has actually happened is that the value in AL has been converted to a binary value. This technique for ASCII to binary conversion works for single digit numeric entries in the range of 0 to 9. When there is more than one digit to convert or the data entered is not a numeric digit, other techniques must be employed—I'll discuss those in a moment.

The next few statements following the label RESPONSE determine which conversion routine is required. Register AL is decremented by one, using the DEC instruction. If the remaining value is zero, program control is transferred to the appropriate routine. The program restarts if the value in AL is greater than 3 (an invalid user response).

NUMBERSY.ASM does not check to see if the character typed by the user was a 1, 2, or 3. You could have entered A, B, or C (41H, 42H or 43H), in which case the value obtained by the masking of the high order bits would also have produced a binary value in AL of 1, 2, or 3.

In other programs the user's response may have to be limited to a certain range or to specific characters. The more error trapping included in a program, the more foolproof the program will be.

This leads to one of my laws about programming: You can make programs foolproof but not idiot proof. There will always be someone who can find the one weak link in a program. If there is not a weak link in the program, then the true "idiot" will spill coffee on program disk! The way around this is to act as your own "idiot" tester. Do everything except spill coffee on the disk. Enter invalid data and see how your program traps the errors. Have one of your friends or neighbors run the program. Test every routine and every module in the program. Use the debugging facility of MS-DOS to single step through a procedure. Set up the registers and data areas with test data that can be checked for erroneous results. Once you've put the program through the acid test of self-examination and extensive testing, your product is ready to be released.

Program Logic Flow

I will now discuss the program logic for the first choice presented to the user: decimal to binary and hexadecimal conversion. The other conversion routines follow similar logic and can be followed by using the flow charts in Figure 7-1 or by inspecting Listing 7-1 Appendix D. As each procedure is discussed, you should look up the procedure in the listing and follow the the source code along with the discussion.

Decimal to Binary and Hexadecimal Conversion

If you answer the initial prompt for a decimal to binary and hexadecimal conversion, the program branches to the label BIN_DEC_HEX_1, which in turn calls the procedure to accept a decimal number from the keyboard.

Procedure: DEC_BIN_HEX

This procedure prompts you for the decimal number to be converted, translates the number to the other numbering systems, and displays the conversions. The procedure begins by setting the MAX_CHARS memory location to 6. This allows you to use the MS-DOS function to accept a line of input from the keyboard. As mentioned earlier, the line input must be terminated by a carriage return. Therefore, the maximum number of characters is set to 6: 5 for the decimal number and 1 for the carriage return.

Program Label: D_B_H_1

Once you have entered the 5-digit decimal number, the string is moved from the keyboard buffer K_BUFF to the storage location DECIMAL. Registers SI and DI establish the source and destination address pointers required for the MOVSB instruction. MOVSB moves a byte of data from the location pointed to by DS:SI to the memory location pointed to by ES:DI. As ES and DS are set to point to the base of the same segment (data segment), MOVSB moves bytes from one part of the segment (K_BUFF) to another (DECIMAL).

Notice that a count of 5 is moved into the CX register, which enables the MOVSB to be repeated using the REP prefix. Each time the instruction is executed, the CX register is decremented by 1. If the CX register does not contain zero, the instruction is repeated. The net result of the operation is that the first 5 digits are moved from [SI] to [DI].

Program Label: D_B_H_2

A call is made to the procedure DECIMAL_BINARY, which performs the necessary decimal to binary conversion. Another call is then made to the procedure BINARY_HEXADECIMAL, which converts the 16-bit binary value stored at BINARY to a 4-digit ASCII encoded hexadecimal value.

Program Label: D_B_H_3

Once the value has been converted to binary and hexadecimal, the results of the conversions must be displayed. Since the storage locations DECIMAL, BINARY_ASCII, and HEXADECIMAL are terminated by 24H ($), the macro @VDLINE can be used to display their contents on the video display. The procedure DISP_ASCII_BINARY displays a binary value on the screen. The procedure DEC_BIN_HEX then returns to the main program at the statement following the DEC_BIN_HEX1 label.

The program again branches, this time to the label DONE_IT_2, which calls two more procedures, FLUSH_BUFFERS and ANOTHER_ONE. FLUSH_BUFFERS clears the keyboard buffer K_BUFF and the RAM locations BINARY, BINARY_ASCII, HEXADECIMAL, and DECIMAL. The buffers are cleared to insure a known state, should the program be repeated. A common mistake in programming is neglecting to reinitialize a data area before it is reused. For the most part, this does not create a problem. In a few instances it can. It is always better to be safe than sorry, so remember to reset buffers, storage locations, and pointers after they have been used and are no longer of importance.

The procedure ANOTHER_ONE simply asks if you want to run the program again and perform another conversion. If you enter an affirmative response, the procedure returns to the calling program with the carry bit set in the flag register. If you enter a negative response, the routine returns with the carry bit reset, and the program terminates.

Decimal to Binary Conversion

Procedure: DECIMAL_BINARY

The procedure DECIMAL_BINARY expects a 5-digit ASCII string to be stored in RAM starting at the label DECIMAL. The routine further expects that each digit is an ASCII value between 30H and 39H,or 0–9. It would make little sense for a decimal number to contain anything but 0–9, yet the routine does not check for the digit's validity. This would be an excellent opportunity for you to write a bounds-checking routine that would parse the string stored at DECIMAL for proper values.

The routine fetches an ASCII digit from memory, masks the high order bits, and multiplies the binary number 0–9 by the decimal power for the digit's position. This was demonstrated earlier in this chapter. The result is then added to the RAM location BINARY. The process is repeated for all five ASCII decimal digits.

The algorithm is implemented as follows: SI and DI are used as pointers to the decimal ASCII string entered by the user and the table DECIMAL_POWERS, respectively. SI and DI are set to zero via the XOR statement, and the RAM location BINARY is set to zero. BINARY must be cleared at the beginning of the conversion routine, as it is used to accumulate the result of the decimal to binary conversion.

A value of 5 is moved into the CX register, which defines the number of times the conversion loop CONV_BIN is executed—once for each decimal digit.

The value stored at DECIMAL + [SI] is moved to the AL register. The ASCII digit is stripped to its binary equivalent and multiplied by the decimal power pointed to by DI. The result is then added to the storage location BINARY. SI and DI are then adjusted to point to the next decimal entry and the next decimal power. Notice that DI must be incremented twice, since the decimal powers are stored as word (2-byte) values. The loop is then repeated for the next digit. When all 5 digits have been converted, the routine returns to the calling procedure.

Hexadecimal to Binary Conversion

Procedure: HEX_BIN_DEC

This procedure converts a 4-digit hexadecimal number stored at HEXADECIMAL to its binary equivalent. It is similar to the decimal to binary conversion routine. The exception is that when a hexadecimal digit in the range of A to F is entered from the keyboard, the digit is represented in ASCII as 41H to 46H. Simply masking the high order bits will not convert the ASCII digit to binary, as it will for the ASCII digits 0 - 9.

What must be done is to first determine if the value is greater than or equal to 41H. If not, the entry is assumed to be in the range of 31H–39H, which is 0 - 9, and it does not require any special consideration when converting the value to binary.

If the digit is equal to or greater than 41H, the digit entered is assumed to be in the range of A - F. The procedure handles this special case with the programming statements found at the label MULT_HEX. To convert the digit to binary, subtract 7 from the digit's value and mask the high order bits. For example:

```
Digit to convert: 41H = 'A'     =  0100  0001  (Binary)

Subtract 7 ---------------->    -  0000  0111  (Binary) ----------
                                   ------------
Result -------------------->       0011  1010  (Binary)

Mask high order bits:  AND         0000  1111  (Binary mask)

Result -------------------->       0000  1010  (Binary)
```

The value has now been correctly converted from base 16 (hex) to base 2 (binary). The next step is to multiply the value by the hexadecimal power for that digit's position in the string. If the digit is the most significant digit, the value is multiplied by 4096. The binary product is accumulated in the location BINARY. The HEXADECIMAL_POWERS table contains the powers for each hex digit's position.

BX and SI are adjusted to point to the next hex digit and the next hexadecimal power. The loop MULT_HEX is then repeated for all four hexadecimal digits. When all the numbers have been converted, the procedure returns to the calling routine.

The conversion assumes that the hexadecimal number which has been entered is 4 digits and in the range of 0 to F. There is no error-checking routine to test whether or not each digit is actually in this range.A good practice exercise for you would be to write a routine that would check for a valid entry. If the character entered is out of range, have the program branch to an error handler routine which would display an error message and ask the user to reenter the data. Hint: The routine, CHECK_BOUNDS, which is part of the program DIRREAD.ASM discussed in Chapter 8, will perform what is required.

ASCII To Binary Conversion

Procedure: ASCII_BINARY_CONV

When the type of conversion selected is binary to decimal and hexadecimal, the data entered will be a 16-byte ASCII string of ones and zeros representing a 16-bit binary value. The procedure's documentation contains the algorithm used in the conversion and is listed in Listing 7-1.

Binary to Decimal Conversion

Procedure: BINARY_DECIMAL

Once a binary value is obtained by using any of the three binary conversion procedures just discussed, a conversion can be made to any of the remaining number bases. So the first step in any of the conversion procedures is to convert the ASCII values to binary, and then convert the binary value to the other number bases. If this were not done, specialty routines would have to be written to convert decimal to hexadecimal, and hexadecimal to decimal.

The algorithm used to convert a binary number to a decimal string begins by dividing the binary value by the higher decimal power, converting the result in AL to an ASCII digit, and storing the digit. Any remainder is recovered and the next highest decimal power is divided into the remaining binary value. The process is repeated until a 5-character ASCII string is built in the RAM storage location DECIMAL.

The procedure begins by clearing the 5-byte storage location at the label DECIMAL in the data segment. A space character (ASCII 20H) is stored in each of the 5 bytes. Next, DI and SI are set to zero and used as pointers to the DECIMAL storage area and the DECIMAL_POWERS table, respectively.

The binary value is moved from memory (BINARY) to AX. The DX register is cleared to accommodate the DIV instruction, which divides the decimal power pointed to by SI into the value contained in AX. The quotient in AL (actually AX) is then logically ORed with 30H to yield a displayable ASCII character. The contents of register AL are stored in the DECIMAL string area at the byte pointed to by DI.

The pointers are adjusted to point to the next storage location for the next decimal digit and to the next decimal power. The remainder from a division is stored in DX following the DIV instruction. Therefore, the remainder in DX is placed in AX via the XCHG instruction, and the loop MAIN_CONV is repeated until all five decimal digits have been constructed. The result is a 5-digit ASCII string that represents the decimal value of the binary word stored at BINARY.

Binary to Hexadecimal Conversion

Procedure: BINARY_HEXADECIMAL

The algorithm used to convert binary numbers to hexadecimal is a little more involved than the binary to decimal routine. The 16-bit binary number must be converted to 4 hexadecimal digits. The following example illustrates the conversion process:

```
Assume the binary word to convert is:

0 1 1 0 1 1 0 1 0 0 0 0 1 0 1 0  (Binary)

1) Take the high order byte of the binary word:

   0 1 1 0 1 1 0 1

   A) Mask the low order nibble of the byte;

      0 1 1 0 0 0 0 0  (60 hex)

   B) Shift the high order nibble left four bit positions
      to the low order nibble.

      0 0 0 0 0 1 1 0

   C) If the value is greater than or equal to 0AH
      add 37H to the value; otherwise, logically OR
      30H with the byte.

      The value is less than 0AH, so we OR 30H with the value.

      0 0 1 1 0 1 1 0  (36H = ASCII `6')
```

continued

```
      D) Store the ASCII-Hex digit.

      E) Retrieve the first byte of the binary word.

         0  1  1  0  1  1  0  1

      F) Mask the high order nibble of the word.

         0  0  0  0  1  1  0  1

      G) Repeat step 1.C for this value.

         Since the value in our example is greater than
         0AH, we must add 37H to the value.

         0  0  0  0  1  1  0  1

       + 0  0  1  1  0  1  1  1
         ----------------------
         0  1  0  0  0  1  0  0  (44 hex)

         The value is now an ASCII D, which is the value
         of the low order nibble of the first byte in
         hexadecimal.

      H) Store the result in the next storage location.

   2) Take the low order byte of the binary word.

      0  0  0  0  1  0  1  0

      A) Repeat steps 1.A - 1.H above for this byte value.
```

In this case, the algorithm is so close to the actual code used in its implementation that I will not discuss each instruction in the routine. Rather, I direct your attention to the procedure as illustrated in Listing 7-1 in Appendix D.

Displaying ASCII-Binary Values

Procedure: DISP_ASCII_BINARY

This procedure displays a binary value stored in the RAM location BINARY. Each bit of the binary word is rotated right 1 bit position via the ROR instruction. By using the ROR instruction, the bit rotated out of the MSB (bit 15) is copied into the carry. This information can be used by a conditional jump instruction. If the bit is 0, the carry is clear. If the bit is 1, the carry is set. The conditional jump instruction allows branching to the portion of the program that places the proper ASCII character (1 or 0) in the 16-byte buffer beginning at label BINARY_ASCII in the data segment.

In each case, the program must display the ASCII equivalent character for that particular bit position (31H for a 1 bit and 30H for a zero bit).

Displaying the String

Once the string has been built in the RAM locations BINARY_ASCII[0] through BINARY_ASCII[15], the program displays each character one at a time with a space character (20H) between the digits. The space is inserted between the ASCII digits to make the video display easier to read.

In Conclusion

The number conversion algorithms are useful general purpose routines that can be incorporated in other programs. For example, it is sometimes convenient to display the register contents of the 8088 when debugging a program. The binary to hexadecimal routine could be altered to perform such a function. Code conversions (such as ASCII to Baudot) and number conversions are some of the most common routines found in programs. These routines and the documentation found in the source program discussed in this chapter should enable you to begin working on more elaborate programs.

8

Disk I/O Programming

One of the most fascinating and intimidating programming chores for novice programmers is writing programs that read from and write to disk. It really isn't difficult to write your own routines utilizing disk I/O, especially when you use the function calls supplied by MS-DOS. Any function accessed via the INT 21H interrupt, can be found in the listing of MACFLE.MAC discussed in Chapter 6 (see Listing 6-1, Appendix D). Each function is well-documented in the listing; therefore, the conventions used in register setup, error reporting, and so on, are not repeated. Consult the MACFLE.MAC listing for the details on the functions used. The disk functions supported by MS-DOS interrupt type 21H are summarized in Table 8-1. Those marked by an asterisk are found only under version 2.0.

Table 8-1
MSDOS Disk Functions
Used with INT 21H

Function Number	Function
0DH	Disk Reset
0EH	Select Disk
0FH	Open a File
10H	Close a file
11H	Search Directory for a file.
12H	Search Directory for the next matching entry.
13H	Delete the specified file from the directory.
14H	Sequential record read.

Table 8-1
MSDOS Disk Functions
Used with INT 21H

Function Number	Function
15H	Sequential record write.
16H	Create a file.
17H	Rename a file.
19H	Fetch the current disk number.
1AH	Set the disk transfer address.
1BH	Get the FAT byte for version 1.x disk (See function 36H)
21H	Random record read.
22H	Random record write.
23H	Get the file size.
24H	Set the random record field (bytes 33-36) in the specified FCB.
27H	Block read.
28H	Block write.
29H	Parse a string for a filename.
2EH	Verify after disk write.
2FH *	Fetch the address of the current DTA.
30H *	Fetch the current MSDOS version number.
36H *	Read the free space available on the specified drive.
39H *	Create a sub-directory.
3AH *	Remove a directory entry.
3BH*	Change the current directory.
3CH *	Create a file.
3DH *	Open a file.
3EH *	Close a file.
3FH *	Read characters from a file or device.
40H *	Write characters to a file or device.
41H *	Delete a directory entry.
42H *	Move address pointer of file.
43H *	Read or change a file's attributes.
45H *	Duplicate a file handle.
46H *	Point a file handle to a new file.
47H *	Read current directory path.
4BH *	Load and execute a program.
4CH *	Terminate current process and close any open files.
4EH *	Find matching pathname with the attributes specified.
4FH *	Find the next matching pathname with the attributes specified.
54H *	Fetch the status of the verify flag (verify/write flag).
56H *	Rename a file.
57H *	Get or set the date time stamp of the specified file.

Before I discuss the routines that access files and the directory using MS-DOS, I want to discuss the structure of the diskette itself. It is important to understand the physical layout of the diskette when using the more advanced disk I/O programming techniques.

The Diskette

The diskette is a digital recording medium that can be thought of as a circular piece of magnetic recording tape. The medium stores magnetic flux reversals, which correspond to a 1 or 0 bit of data. The diskette is divided into tracks, sectors, and clusters. There are 40 tracks on a diskette, as formatted with the IBM PC. Other MS-DOS machines use disk drives capable of formatting 80 tracks on a diskette.

Figure 8-1 illustrates the layout of a diskette. The outermost track is track 0 and the innermost track is track 39. Track and sector numbering always begins with zero rather than one. Each track is divided into either 8 (MS-DOS 1.x) or 9 sectors per track. (MS-DOS 2.0). You can also read and write 8 sectors per track under MS-DOS 2.0, allowing you to read and write version 1.x formats. Each sector is capable of storing 512 bytes of data.

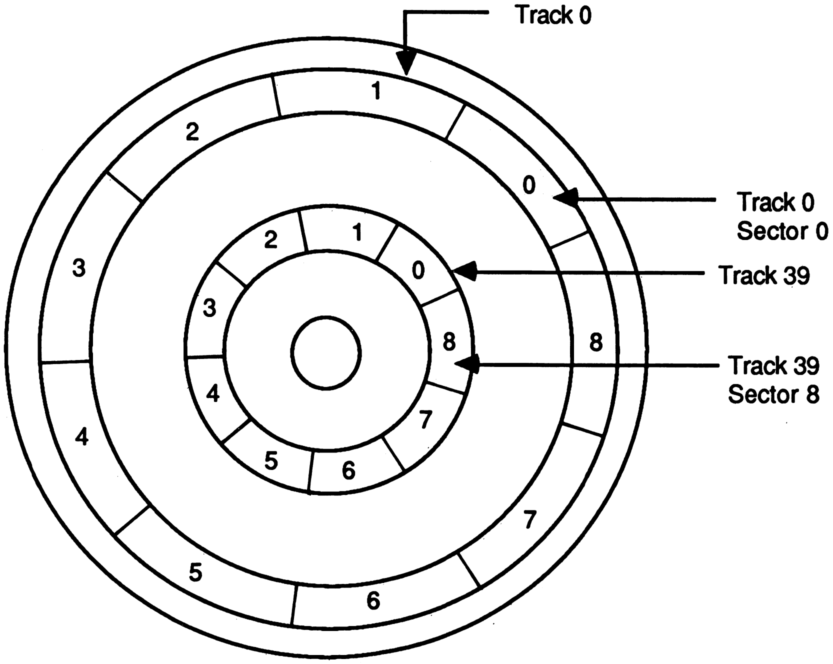


Figure 8-1

Disk Drive Differences

To confuse matters more, not all drives are the same. Some disk drives contain only one read and write head and are referred to as single sided disk drives. When the drive contains two read and write heads, it is referred to as a double sided drive. Single sided disk drives are capable of reading and writing information on one side of the diskette, while a double sided drive can store twice as much information, as both sides of the diskette can be written to.

Single Density versus Double Density

Another issue is the recording technique employed to encode data on the diskette. When a technique called Frequency Modulation, or FM, is employed, the recording technique is referred to as a single density recording format. When MFM, or Modified Frequency Modulation, is used as the data encoding technique, it is referred to as a double density recording technique. The recording format is determined by the disk controller and not the disk drive, as most drives being manufactured today are capable of recording in either single or double density. The issue is not a major one from an application programmer's standpoint, yet the difference is important from the standpoint of understanding how the diskette is organized. Like the single sided versus double sided issue, double density recording of data allows twice the data storage of single density recording. All drive types used with the IBM PC record data using the MFM recording format (double density).

To calculate the total storage capacity of a 5 1/4-inch diskette as formatted under MS-DOS, you need to know whether the drive is single or double sided, the total number of tracks available, and whether there are 8 or 9 sectors per track. Table 8-2 summarizes the formatted storage capacities for the different types of IBM disk formats.

Clusters

IBM PC-DOS/MS-DOS introduces a concept known as a cluster; this is either one sector for single sided diskettes or two adjacent sectors for double sided diskettes. The use of clusters in referring to sector numbers in a file is a result of trying to minimize read and write head movement during disk access.

For single sided MS-DOS version 1.x and 2.0 diskettes, the sectors are numbered from 0 to 319. Single sided disks are allocated 1 sector per cluster. For MS-DOS 2.0 double sided diskettes, the sectors are numbered from 0 to 7, starting with head 0, 0 track 0, and from 8 to 15, starting with head 1, track 0. The *IBM Disk Operating Manual* states that all available sectors are used before moving on to the next track.

Table 8-2
Diskette Capacities

Data Capacity in bytes Single Sided 8 Sectors per Track:

Version 1.0	Version 1.1	Version 2.0
160,256 (SS8ST)	160,256 (SS8ST)	160,256 (SS8ST)
147,968 *	146,434 *	119,296 *

Data Capacity in bytes: Double Sided 8 Sectors per Track:

Version 1.1	Version 2.0
322,560 bytes	322,560 bytes
308,736 bytes	282,600 bytes *

Data Capacity in bytes: 9 Sectors per Track
Version 2.0 Only.

Single Sided	Double Sided
179,712 bytes	362,496 bytes
138,752 bytes *	321,536 bytes *

*Capacity when diskette is formatted with the /s option which writes the IBMBIO.COM, IBMDOS.COM, and COMMAND.COM files to the disk. See the disk map in Table 8-3.

Therefore, data are written to or read from disk using the lower numbered head first (head 0) for all sectors (0–7). Then all sector numbers (8–15) are used for the same track using the other head (head 1). Once the total number of sectors allocated to a track has been utilized, the head advances to the next track. For double sided disks, two consecutive sectors are allocated to each cluster.

MS-DOS version 1.1 double sided sector numbers begin with head 0, track 0, sectors 0–7. Unlike the 2.0 format, head 1, track 0 begins its numbering at sector number 320. Sector numbers are allocated sequentially on the first side of the disk from track 0, head 0, sector 0 through track 39, head 0, sector 319. Then the sector numbers continue sequentially from the opposite side of the disk from track 0, head 1, track 320 to track 39, head 1, sector 639.

MS-DOS maps the cluster allocation via the file allocation table (FAT). The clusters inform MS-DOS what clusters have been allocated to a given file. Page 4-10 in the *IBM DOS Technical Reference Manual* contains the information needed to convert clusters to actual sector numbers for a file. Most applications do not require an in-depth knowledge of clusters and file allocation space. It does become important if for some reason you need to directly access a sector for a file.

Table 8-3 illustrates the numbering sequence for sectors under MS-DOS 1.x and 2.0 for the various formats found in each version. The disk map in Table 8-3 also shows which sectors and tracks are allocated to specific system files. The files marked with an asterisk exist whenever you have formatted a new diskette using the system option (/s); otherwise, the sectors normally occupied by the files are available for user data.

Table 8-3
MSDOS Disk Map
Version 1.0/2.0/2.10

Single-Sided 8 Sectors/Track			Double Sided 8 Sectors/Track			
File	**Track**	**Sectors**	**File**	**Track**	**HD**	**Sectors**
BOOT	0	0	BOOT	0	0	0
FAT	0	1-2	FAT	0	0	1-2
DIREC	0	3-6	DIREC	0	0	3-7
				0	1	320-321
BIOS *	0	7	BIOS *	0	1	322-325
	1	8-10				
MSDOS *	1	11-15	MSDOS *	0	1	326-327
	2	16-23		1	0	8-15
				1	1	328-330
COMMAND *	3	24-31	COMMAND *	1	1	330-335
	4	32-33		2	0	16-20
User files begin on;						
Track 4, Sector 34			Track 2, Head 0, Sector 21.			
And Continue Until:						
Track 39, Sector 319			Track 39, Head 1, Sector 639			

*IBMBIO.COM, IBMDOS.COM, COMMAND.COM exist when the diskette is formatted with the /s option.

Version 2.0

Single Sided 9 Sectors/Track			Double Sided 9 Sectors/Track			
File	**Track**	**Sector**	**File**	**Track**	**Head**	**Sector**
BOOT	0	0	BOOT	0	0	0
FAT	0	1-4	FAT	0	0	1-4
DIREC	0	5-8	DIREC	0	0	5-8
				0	1	9-12
BIOS *	1	9-17	BIOS *	0	1	13-18
	2	18		1	0	19-22
MSDOS *	2	19-26	MSDOS *	1	0	23-27
	3	27-35		1	1	28-36
	4	36-44		2	0	37-45
	5	45-52		2	1	46-54
				3	0	55-56
COMMAND *	5	53		3	0	57-63
	6	54-62		3	1	64-72
	7	63-71		4	0	73-81
	8	72-80		4	1	82-90
	9	81-88		5	0	91-92
User Files Start On:						
Track 9, Sector 88			Track 5, Head 0, Sector 3			
And Continue Until:						
Track 39, Sector 359			Track 39, Head 1, Sector 719			

*IBMBIO.COM, IBMDOS.COM, COMMAND.COMM exist only if the diskette was formatted with the /s option.

MS-DOS Programming Tools

The File Allocation Table (FAT)

MS-DOS provides the applications programmer with a complete set of programming tools via the function calls it provides for disk access. You do not need to know whether the drive you are working with is single or double sided, or has 8 or 9 sectors per track. MS-DOS uses a descriptor byte in the FAT (file allocation table) to determine the format of the diskette being used.

The FAT maintains a correct mapping of the available and used space on the diskette. The FAT occupies one sector under MS-DOS 1.x and two sectors (sector 1 and 2) under MS-DOS 2.0 and starts at track 0, logical sector 1 for all versions. A copy of the FAT is maintained at track 0, sector 2 for MS-DOS 1.x and at sectors 3 and 4 for version 2.0. Each entry consists of three hexadecimal entries that inform MS-DOS as to whether a cluster in use (000) is the last cluster of a file (FF8–FFF) or the next cluster number in a file (any other 3-digit hex number).

When you format a diskette and bad sectors, or sectors which cannot be formatted, are found, MS-DOS writes an FF7H in the FAT entry for that cluster to indicate that the cluster is bad. Furthermore, the first two entries in the FAT map the directory. The first byte of the first entry indicates the format of the diskette as follows:

```
FAT Byte 0 = FF = Double sided, 8 sectors per track.

             FE = Single sided, 8 sectors per track.

             FD = Double sided, 9 sectors per track.

             FC = Single sided, 9 sectors per track.

             F8 = Hard disk.
```

The remaining two bytes of the first FAT entry are always FFFFH.

This information is not only useful to MS-DOS, which must manage different disk formats, but you can make use of this information as well. A little later in this chapter, I'll demonstrate how to use the media descriptor byte of the FAT in a program that reads and sorts the directory, regardless of the diskette's format.

As mentioned previously, page 4-10 in the IBM disk operating system reference manual contains the information necessary to convert cluster numbers into sector numbers. Only in the most demanding programming applications will you need to be concerned with the cluster numbers as depicted in the FAT entries.

The MS-DOS Directory Structure

The directory for MS-DOS diskettes is found on different sectors and different tracks for each of the different versions of MS-DOS. All versions begin the directory on track 0. Table 8-4 summarizes the differences between the different diskette formats and where the directory can be found.

Table 8-4
Where to find the Directory

Disk Format	Beginning Sector	Total Sectors	Entries
SS8ST	3	4	64
DS8ST	3	7	112
SS9ST	5	4	64
DS9ST	5	7	112

Key: SS8ST = Single Sided 8 Sectors/Track
DS8ST = Double Sided 8 Sectors/Track
SS9ST = Single Sided 9 Sectors/Track
DS9ST = Double Sided 9 Sectors/Track

Notice that the single sided formats allow a total of only 64 directory entries per disk, as opposed to 112 possible directory entries for the double sided formats.

Each byte of a directory entry has a particular significance. The byte significance found in a directory entry is summarized in the listing of the directory read/sort program discussed at the end of this chapter and shown in Table 8-5.

Working with MS-DOS Disk Functions

MS-DOS provides a variety of disk functions to operate on files, directory entries, and absolute sectors and tracks. Examples of how the functions are used in a program are given in the macro file listing discussed in Chapter 6 (Listing 6-1, Appendix D). Use the functions as macros or adapt them as in line code for use in your programs. Version 2.0 introduces many new concepts and high level disk

Table 8-5
Format of the Directory

Byte(s)	
0-7	Filename. If byte 0 = 00H then entry never used. E5H then entry has been deleted 2EH then the entry is for a directory Any other character found in byte 0 is the first character of the filename.
8-10	The file name extension.
11	The File attribute 00 = Normal File 01 = Read Only * 02 = Hidden File 04 = System File 08 = Volume Label (in bytes 0-10) * 10 = Subdirectory * 20 = Archive Bit, set whenever the file has been written to and closed, and has not been backed up.
12-21	Reserved by MSDOS
22-23	The time the file was last updated or created. Format: B11-15 = Hour (0-23) B5-10 = Minutes (0-59) B0-4 = two-second increments Least significant byte = Byte 22 Most Significant Byte = Byte 23
24-25	Date of file creation or update. Format: B8-15 = Year (0-119 = 1980-2099) B5-7 = Month (1-12) B4-0 = Day (1-31)
26-27	Starting Cluster; See IBM DOS Technical Reference Manual Page 4-7 and Page 4-10.
28-31	File Size in bytes (Least Significant Word = LSB of the size)

*These attributes are available under MSDOS 2.0 only. Version 2.0 contains all the attribute types from 1.0, and adds these new attribute types.

functions that are not found under MS-DOS 1.x. You can find the functions listed by function number in Table 8-1. Those marked with an asterisk are available only with MS-DOS 2.0.

The Program Segment Prefix

I have postponed discussing the PSP as you did not have to know about it or use it in the programming examples. However, when working with the disk functions available under MS-DOS it is important to understand the PSP's function and organization. Table 8-6 summarizes the PSP.

Table 8-6
The Program Segment Prefix

Byte(s)	Function
0 - 1	0CDH 20H = INT 20H
2 - 3	Memory Size (0010H = 64K, 0020H = 128K, 0040H = 256K)
4	Reserved by MSDOS
5	9AH = Long Call to Function Dispatcher.
6 - 7	Offset address of dispatcher (IP)
8 - 9	Segment address of dispatcher (CS)
0AH - 0BH	Offset Terminate Address (IP)
0CH - 0DH	Segment Terminate Address (CS)
0EH - 0FH	Offset Control Break Exit Address (IP)
10H - 11H	Segment Control Break Address (CS)
12H - 13H	Offset Critical Error Exit Address (IP)
14H - 15H	Segment Critical Error Exit Address (CS)
16H - 5BH	Reserved by MSDOS (2CH contains the segment for the environment)
5CH - 6BH (or 80H)	Formatted Parameter Area 1. Formatted as an unopened FCB.
6BH - 7FH	Formatted Parameter Area 2.
80H - 0FFH	Unformatted parameter area, acts as the default Disk Transfer Area.
100H -	Start of User's Program.

Each time a program is loaded and executed from disk, MS-DOS creates a PSP. It is 256 bytes in length and extends from offset 0 to 100H of the segment where your program resides. The first byte of your program is therefore at 100H in the segment. The PSP can be used by the program and by MS-DOS to pass parameters to each other. As an example, let's say you want to run a program, but you want to load a different data file for use in the main program each time the program is run. You could type the following command from the MS-DOS prompt:

```
A> COSTPROG.EXE DATA_1.FLE
```

Here you have commanded MS-DOS to run the program COSTPROG.EXE and passed a parameter to the program (DATA_1.FLE). The MS-DOS program loader copies the parameter specified on the command line into the PSP at byte 81H. The length of the parameter is copied to the PSP offset byte 80H. A copy of the parameter is also placed in the PSP at offset 5CH, which is used by MS-DOS as the default file control block (more on FCBs in a moment).

The program can examine the PSP at location 80H for the length of a parameter string and find the parameter string (if any), beginning at offset 81H in the PSP. Notice that the first 2 bytes of the PSP consist of an INT 20H instruction. The INT

20H instruction restores the MS-DOS environment when at the end of the program. It is the normal manner in which user programs are expected to terminate.

By now you should be accustomed to the fact that every program you write begins by pushing the DS register on the stack, followed by a word of zeros. When you exit a program via the RET instruction, as is the case in most of the examples thus far, a FAR return is executed and register IP is loaded with the word of zeros that was stored on the stack. CS is loaded with the segment of the PSP which was pushed on the stack at the start of the program. These values form a pointer to the first byte of the PSP. After executing the RET instruction, the INT 20H instruction at offset 00H in the PSP is executed.

The net result is that all logic, pointers, and file buffers are restored to the environment that existed prior to loading and executing the program.

File Control Blocks

When operating on disk files using version 1.x, you need to know what a file control block (FCB) is and how it is organized. The FCB contains information about the file: the drive number, the filename and extension, the date of creation, and pointers into the file.

An FCB can take on one of two possible forms, normal or extended. The extended FCB contains a 7-byte prefix containing the attribute of the file, as summarized in Table 8-5. Version 2.0 supports more possible file attributes than does version 1.x. Those attributes found in the table and marked with an asterisk are found under version 2.0 only. Normal FCBs are 37 bytes in length, while an extended FCB is a total of 44 bytes in length.

In the previous discussion about the PSP, I mentioned that MS-DOS places the first parameter specified from a command line in the PSP as a formatted FCB (PSP offset 5CH). MS-DOS uses this as the default location for the FCB during disk I/O unless instructed otherwise. If a second parameter is issued from the command line, it is also copied into the PSP, but at offset 6CH. You may have noticed that 6CH is neither 37 bytes nor is it 44 bytes from 5CH. Consequently, if you use the default FCB at 5CH in the PSP, the second FCB is overwritten and destroyed.

You can avoid all of this by setting aside memory in your data segment for the FCBs used in your program. If you need to have two files open at the same time in your program, define two FCB's in your data segment. If you only have one file open at a given time, you can get by with only one FCB. If you intend to pass parameters to your program that are actually filenames, move them from the PSP default areas 5CH and/or 6CH, to the memory you reserved in your program's data segment for FCBs.

The FCB bytes you must be concerned about when accessing files under MS-DOS are denoted in Table 8-7 with an asterisk. You must set up the drive number, filename and extension, current block number, and logical record size. The MS-DOS access functions expect these fields to be initialized. When opening or creating a file, MS-DOS initializes the record size to 128 bytes. Should you want to operate on record sizes greater than or less than this default, it is necessary to change the record size for your application.

Table 8-7
File Control Block Formats

Byte(s)	Function
FCB - 7	0FFH = Denotes an extended FCB.
FCB -6 through -2	00H
FCB -1	Attribute Byte (See Table 8-5)
FCB + 0 *	Drive number 0 = default 1 = A 2 = B After Open: 1 = A, 2 = B
FCB + (1 thru 8) *	Filename. Left justified with trailing blanks.
FCB + (9 thru 0BH) *	Extension for filename, left justified, with trailing blanks (or all blanks).
FCB + (0CH thru 0DH) *	Current block number relative to the beginning of a file. Each block consists of 128 records, each with a size as specified by the word at 0EH. This field is used with the current record field to perform sequential read operations. A value of zero denotes the first 128 records (first block) of the file. Initialized to zero by MSDOS.
FCB + (0EH thru 0FH) *	Logical record size. The logical record size is specified in bytes. It must be set after the open or the create functions. MSDOS initializes this field to 80H after creating or opening a file.
FCB + (10H thru 13H)	File size in bytes. Initialized by MSDOS from information found in the directory.
FCB + (14H thru 15H)	Date Field. The date the file was created or last updated.
FCB + (16H thru 1FH)	Reserved by MSDOS
FCB + 20H *	Current Relative Record number (0-127), within the current block. This field must be set by the user before performing sequential record access.
FCB + (21H thru 24H) *	Relative record number (also referred to as the random record field). This field must be initialized by you before performing a random R/W function call. You can use the MSDOS function 24H to set this field based on the information in the current block and current record fields. This field holds the relative record number from the beginning of the file for random record or block access.

*User must initialize these fields. Consult the text for an explanation of the FCB.

MS-DOS also initializes the current record number to zero. If you want to start at a record number other than zero, you have to change the value stored in this field to the record number desired.

Specifying the FCB

Format: **MS-DOS Function: 0FH**

Macro: **@OPEN**

When you use the MS-DOS function calls for disk I/O, the FCB for the file you want to use is pointed to by DS:DX. For example, to open a file, you would use the MS-DOS function 0FH. Assuming you have established an FCB in the data segment (let's call it MY_FCB for now), initialize the first 11 bytes as follows:

Byte 0 = Drive number (0,1, or 2),

Bytes 1-8 = File name (8 characters maximum),

Bytes 9-11 = The file name's extension (3 characters maximum).

Later in this chapter, I'll show you how to prompt a user for a drive and filename, then use the MS-DOS PARSE_STRING function 29H to move the string to an unopened FCB. It saves you time and makes life simpler when creating or opening files.

Next, use the following program statements to open the file:

```
LEA DX,MY_FCB          ;Address pointer to the FCB

MOV AH,0FH             ;MSDOS function code

INT 21H                ;Try to open the file.
```

If the file does not exist in the directory, AL will contain 0FFH when control is returned to your program. If AL contains 00H, the entry was found and the file was opened. You can use this information to create a file, if the file specified is not found in the directory. See MS-DOS function 16H in Listing 6-1, Appendix D or the discussion of the function found later in this chapter.

The main reason you want to first use the open function rather than the create function is that, if you attempt to create a file and the file already exists in the directory, MS-DOS will set the file size field to zero. This in effect erases any information previously stored in the existing file. The rule here is to study the listing of the functions in Listing 6-1 before you use them and know what you are doing.

When working with the disk access functions having the potential to erase the contents of a file or a disk, use a disk with data that can easily be replaced. Never experiment on your only copy of a diskette containing valuable data. Now that you have been warned by someone who is an expert at erasing disks, let's move on.

When using the open or create functions, MS-DOS takes the liberty of initializing the current block number (FCB bytes 0CH–0DH) and the record size field (FCB bytes 0EH–0FH) to 0000H and 0080H, respectively. You must change these fields after the file has been opened if they are not suitable for your application. If you plan to begin with the first block of the file (a block is 128 records) and the record size of the file is 128 bytes, then do nothing and leave these fields with the values MS-DOS supplies.

Bytes 10H–16H are also initialized by MS-DOS with the file size and date information found in the directory entry for the file. MS-DOS does not initialize the current record number byte in the FCB (FCB byte 20H). You must initialize this byte after opening or creating a file. Usually you will set the record number to zero to access the first logical record within the current block.

The next step before performing a sequential read/write or random access read-/write operation is to set the disk transfer address.

Disk Transfer Address

Another area that must be established in your data segment prior to using the disk access functions is the Disk Transfer Address, or DTA. The DTA is the address of a buffer used for disk I/O. It can be thought of as the holding area, or buffer, for data that is read from or written to disk. The buffer must be at least as large as the largest record length being used in your program. If your record length is 1 byte, then your buffer must be 1 byte or greater in length. If your record length is 512 bytes, then the buffer must be a minimum of 512 bytes in length.

When writing to disk, the data must be placed in the buffer. Then, by using one of the MS-DOS disk write functions listed in Table 8-1, the data are removed from the buffer and written to disk at the current record number specified in the FCB. Similarly, when data are read from disk, the data are placed in the buffer. Your program is responsible for moving data into and out of the buffer.

MS-DOS 2.0 contains functions that allow you to set or obtain the DTA. The DTA must be set before each write and read operation when multiple files are opened at the same time. If you do not set the DTA, MS-DOS uses the last DTA in effect. It may be a DTA belonging to another file. When a program is initially executed from MS-DOS, not only are the default FCBs built into the PSP but also a default DTA which starts at the PSP offset 80H is established. Remember to set the disk transfer address with the MS-DOS function 1AH prior to using the read and write functions.

Creating a File

Function: **16H**

Macro: **@CREATE_FILE**

The create function tries to locate the specified file in the directory. If found, the function uses the OPEN function to open the file, and then sets the file length field in the FCB to zero. The effect of this action is to erase the file's contents. If the filename specified is not found in the directory, the create function creates a new file, initializes the FCB current block and record fields to zero, and sets the record size field to 0080H. You must change these values after the function has returned control to your program, should they prove to be unsuitable for your application.

To invoke the function, move the MS-DOS function code 16H into AH, point DS:DX to the FCB which contains the drive, filename, and extension, and then issue the INT 21H instruction. On return, AL will contain 00H if an open directory was found and 0FFH if the file could not be created, as an open directory entry was not available.

Methods of File Access

Under MS-DOS there are three principal file access mechanisms: sequential read and write, random record read and write, and random block read and write.

Sequential Access

Sequential access operates on the record pointed to by the current block and record number fields in the FCB. After the record has been written to or read from, the current block and current record numbers are adjusted to point to the next logical record in the file.

When the current record field is incremented past 128 (the total number of records within a block), the current block number is incremented to point to the next 128 records in the file. It's a little like the way we tell time. We know there are 60 seconds to the minute, 60 minutes to the hour, and 24 hours to a day. When we exceed 60 seconds, the minute counter of the clock is incremented.

The access logic used by MS-DOS is as follows: There are a certain number of bytes to a record (determined by the logical record size field in the FCB), 128 records to a block, and an indeterminate number of blocks to a file. Since the current block field is a 2-byte field, the maximum number of blocks that theoretically could be present in a file is limited to 65,536 blocks, each containing 128 records.

I admit I have not pushed MS-DOS to such an extreme; yet it gives you some idea of the amount of data MS-DOS is capable of accessing. By setting the current block and current record fields to a value of your choice, you can begin sequential access anywhere in the file. You do not necessarily have to start at the beginning of the file.

Sequential Read

MS-DOS Function: **14H**

Macro: **@READ_SRECORD**

The sequential read function is invoked by moving the address of the FCB for the file to be read into DS:DX, moving the function code 14H into AH, and executing the INT 21H instruction. The FCB must belong to a file that has been opened previously or created by functions 0FH or 16H. The sequential read function reads the record pointed to by the current block and record number found in the FCB. The logical record size field determines the size of the record that is transferred to the DTA. See MACFLE.MAC (Listing 6-1) in Appendix D for a description of the error codes returned by this function.

Sequential Write

MS-DOS Function: **15H**

Macro: **@WRITE_SRECORD**

This function writes the data from the DTA to disk at the record pointed to by the current block and record fields of the FCB. The number of bytes transferred is determined by the logical record size field of the FCB. To invoke the function, set DS:DX to point to the address of an FCB for the open file, set AH to 15H, and execute the INT 21H instruction. See the description of this function in Listing 6-1 in Appendix D for a description of the possible error codes returned.

The IBM DOS reference manual states that, should the record size be smaller than a sector (512 bytes), the data will be transferred from the DTA to a buffer. The buffer is written to disk when the buffer is filled, when the file is closed, or when a disk reset command is issued (MS-DOS function 0DH).

I have had luck with this function only when the record size is divided evenly into 512: 1,2,4,8,16,32,64,128,256 or 512. When I used a record length of 94 or 27, which do not divide evenly into 512, the data written to the buffer were not transferred to disk when the file was closed.

Random Record Access

Another method of access available to the AL programmer using MS-DOS function calls is random record access. If you have ever been involved with one of those Saturday afternoon home construction projects, you'll understand what random access is and how it works.

For your construction project you need to find three items: nails, a hammer, and wood. You pick up the telephone book to call around to some of the local hardware stores to see if they have what you need in stock. Now imagine that you had to look for those telephone numbers by reading the phone book from the first page to the page where the numbers are. This would be similar to sequential access.

Random access allows you to open the telephone book to the page that has Joe's Hardware and Supplies listed, and then to the page containing the number for Zebra Nails or YoYo's Hammers. You do not have to read or wade through every entry. This is random access, and it is quite valuable when searching for specific records.

In order to access a record using the random read/write functions available through MS-DOS, the random record field of the FCB (bytes 21H–24H) must be initialized with the record number desired. The record number specified must be in terms relative to the beginning of the file.

The random record field can be set automatically, based on the information contained in the current block and record fields of the FCB. Using the MS-DOS function 24H will set the random field based on the current block and record fields.

Random Record Read

MS-DOS Function: **21H**

Macro: **@READ_RRECORD**

To read a record from an open file using the random record read function, move the address of the FCB to DS:DX, the function code 21H to AH, and execute the INT 21H instruction. On return from the function, the record specified in the random record field of the FCB will have been read from disk into the DTA.

The random record number will not be altered; it will contain the same value as it did prior to executing the random read function. The current block and current record fields will be updated to point to the next sequential record in the file. If the operation was successful, then AL contains 00H. If AL is nonzero an error has occurred. See the MACFLE.MAC listing in Appendix D for a discussion of the error codes returned by this function.

Random Record Write

MS-DOS Function: 22H

Macro: @WRITE_RECORD

This function writes to disk a record of a length specified by the record size field of the FCB. The record written to is specified by the random record number found in the FCB (bytes 33-36). To invoke the function, move the address of the FCB into DS:DX, set AH equal to 22H and execute the INT 21H instruction.

When control is returned to your program, AL contains either 00H (the operation was successful), 01H (disk was full), or 02H (insufficient space in the DTA to write one record). The current block and record numbers are adjusted to point to the next sequential record following the record just written to. The random record field remains unaltered.

Random Block Access

The random block functions enable us to read to or write from disk more than one record at a time. Actually, the records are written to or read from disk in a sequential manner. IBM and Microsoft use the term random very loosely in describing this function. Random block write means that the current block and record numbers used in sequential access are calculated from the random record field in the FCB. They form a pointer to the first record to be manipulated. This may be confusing at first, but it really is not as confusing as it sounds.

The records are part of a block (or blocks) that is read or written in a sequential manner. Therefore, it makes sense that the sequential access fields of the FCB are used during the operation. The main point to remember is that the sequential access fields pertaining to the current block and record are calculated from the random record field.

After either the random block read or the random block write functions have been executed, the current random block and random record fields will point to the record immediately following the last record written.

Random Block Read

MS-DOS Function: **27H**

Macro: **@BLOCK_READ**

This function reads a specified number of records into the DTA. The DTA must be large enough to hold the maximum number of records specified. To invoke the function, move the FCB address for the open file into DS:DX, the number of records to be read into CX, and the MS-DOS random block read function code 27H into AH. Then execute the INT 21H instruction.

On return, if AL contains 00H, the specified number of records has been read into the DTA. If AL contains any other value, the operation was unsuccessful. CX contains the actual number of records read and may be compared to the value attempted. The MACFLE.MAC listing in Appendix D contains additional information on this function.

Random Block Write

MS-DOS Function: **28H**

Macro: **@WRITE_BLOCK**

This function writes a specified number of records to disk, beginning at the record number specified in the random record field of the FCB. To invoke the function, set DS:DX to the address of the open FCB for the file to be written to, specify the number of records to write in CX, set the MS-DOS function code 28H in AH, and execute the instruction INT 21H.

If the number of records specified in CX is zero, the function will not write any records to disk. However, it will alter the file size entry in the directory. The new file size is set to the random record field of the FCB. MS-DOS will automatically allocate or release clusters as required.

Use this function with caution. The first time I used it, I erased half of the file I was working on. I was trying to increase the size of the file, and somehow specified a smaller random record number than the size of the existing file. I set CX to zero, and my new file size was half the size of the old one. I have since come to use this function with a great amount of respect and caution. Remember, I'm an expert at erasing files (I'm getting quite good at restoring them also).

On return from this function, CX contains the actual number of records written. AL contains 00H if the operation was successful and 01H if there was insufficient disk space to write the specified number of records. The current block and record number fields and the random record field is updated to point to the record immediately following the last record written to disk.

Closing a File

MS-DOS Function: **10H**

Macro: **@CLOSE**

This function closes the file specified in the FCB pointed to by DS:DX. Files that have been opened must be closed using this function. Failure to close a file results in the directory not being updated correctly.

On return from the function, AL contains either 00H to indicate the entry was found in the directory or 0FFH, which indicates that the entry was not found in the directory. To invoke the function, move the effective address of the FCB into DS:DX, 10H into AH, and execute INT 21H.

You are directed to the IBM DOS manual and to the macro definitions found in Chapter 6 for a description of the other MS-DOS 1.X disk I/O functions found in Table 8-1.

Methods of File Access Under MS-DOS 2.0

File Handles

MS-DOS 2.0 implements a new concept in disk access—file handles. A file handle is a unique 2-byte number that MS-DOS assigns to a file when it is opened or created, using the MS-DOS functions 3DH (open) or 3CH (create). The file handle is returned in AX if the operation was successful (see MACFLE.MAC in Appendix D). All future references to the file are then made in regards to the file handle. The important point to remember is that you do not need to specify an FCB for any file you want to work with under MS-DOS 2.0. Simply use the file handle when referencing the file.

Furthermore, file access has been greatly simplified from the application programmer's viewpoint with the release of MS-DOS version 2.0. You do not have to worry about file control blocks when accessing files, as MS-DOS creates its own as needed. You still must set the DTA using MS-DOS function 1AH when using the version 2.0 system calls.

Tree Structures

Popular with many advanced operating systems such as Unix (registered trademark of AT&T), tree or hierarchical directory structures allow a diskette to store different files under different directory names. Each directory name may contain subdirectories or files. To access a file in a given directory, you need to know the pathname.

Pathnames

Opening a File

MS-DOS 2.0 Function: **3DH**

Macro: **@OPEN_FILE_2**

Let's say that you maintain multiple mailing lists and have created the directory MAIL. You store mailing lists for work and professional associates and for personal acquaintances. The files have been named WORK and HOME. You can use the MS-DOS 2.0 function 3BH to move to the desired directory. Once you have chosen the proper directory, specify the pathname for the file as:

```
d:\MAIL\WORK    or    d:\MAIL\HOME
```

The primary motivation for the architecture of the directory was to manage the massive amount of information the IBM PC/XT is capable of storing on the hard disk. With 10 megabytes or more available to the user of the more sophisticated machine, the tree structure was essential. There may be only a few different working directories shown on the drive, while multiple files may be stored in each working directory.

MS-DOS 2.0 returns a file handle after opening the file. Use the file handle and not the pathname when accessing the file using the read and write functions provided under MS-DOS 2.0. To open a file using a pathname, use function 3DH. You must also specify the access mode which is either:

0 for read only

1 for write only

or 2 for read/write access

The access mode must be in AL, and DX must point to the ASCIIZ pathname (ASCIIZ is an ASCII pathname terminated with a null byte, 00H). CX must contain the file attribute (see Table 8-5), and AH must contain 3DH. Then execute INT 21H. On return, if the carry is reset, the operation was successful, and the AX register contains the file handle. If the carry is set, then register AX contains an error code as follows:

```
AX = 03 = Pathname not found.

AX = 04 = Too many open files.

AX = 05 = Access denied.
```

After using those functions unique to version 2.0, should the carry be set, an error has occurred. Under version 1.0, several different and inconsistent methods were employed to report errors in using the MS-DOS functions. The different methods for reporting errors are detailed in Appendix D in the listing of MACFLE.MAC. You should also reacquaint yourself with the disk functions listed in Table 8-1.

In most of the programming you will be involved in, the functions provided under MS-DOS 2.0 are superior to those found in version 1.x. If you are still using version 1.0, use the version 1.x macros as supplied in MACFLE.MAC and in the manner prescribed in their documentation. MS-DOS 2.0 is compatible with those functions found in versions 1.0 and 1.1. Therefore, programs written using the disk functions provided under MS-DOS 1.x also work under 2.0.

Creating a File

MS-DOS 2.0 Function: 3CH

Macro: @CREATE_FILE_2

This function is similar to the 1.x function, which creates a file such that if an existing file is specified, its length is set to zero, thereby erasing the file's contents. To invoke the function, move 3CH into AH, point DS:DX to the ASCIIZ pathname, CX to the file attribute (see Table 8-5), and execute interrupt 21H. On return, the carry is reset if the operation successfully created a file and set if there was an error. See the macro listing for this function in MACFLE.MAC for a complete description of the error codes returned.

MS-DOS 2.0 Read/Write Functions

The read/write functions under 2.0 are:

1. Read from a file or device: 3FH,
2. Write to a file or a device: 40H.

The functions are summarized in MACFLE.MAC. The description explains the functions and need not be repeated here.

Closing a File

MS-DOS Function: **3EH**

Macro: **@CLOSE_FILE_2**

As I mentioned in the description of the MS-DOS 1.x CLOSE function, a file that has been opened must be closed to insure a proper update of the directory entry for the file. To close a file that was opened or created using the MS-DOS functions 3DH or 3CH, you must use the 2.0 CLOSE function. Put the file handle for the open file in the BX register and the function code number 3EH into AH, and execute INT 21H. On return, the carry is set if an error occurred. If the carry is set, AL contains the error code 06H indicating that the file handle you supplied to the CLOSE function was not for an open file.

The other MS-DOS functions found under version 2.0 are detailed in the MACFLE.MAC. I have repeated the most frequently used functions in this chapter, only to preface the programming examples you are about to encounter. You should take time to study those functions listed in Table 8-1 and read the description of each function as given in Chapter 6. The MACFLE listing is quite complete and provides you with all the information necessary to use the functions.

Absolute Disk Access

When it is necessary to write disk routines and procedures that directly access a given track and sector, two methods of access are available. The first involves using the MS-DOS interrupt types 25H (disk read) and 26H (disk write). These are not function numbers to be used with INT 21H. They are interrupt types. Use INT 25H for absolute disk access for read operations and INT 26H for absolute disk access for write operations.

The other option is to use the BIOS interrupt 13H for absolute disk access. Use of the BIOS interrupt may result in your application program not running on another MS-DOS machine due to BIOS differences. The listing MACFLE.MAC contains this interrupt function and other BIOS routines.

Logical versus Physical Sectors

Logical sectors are numbered one less than the physical or absolute sector number. For example, the logical sector for track 0, sector 1 is 0. The logical sector for track 10, sector 1 on a single sided disk with 9 sectors per track is 90. Since there are 9 sectors per track and we are interested in the first sector of track 10, the logical sector number is 90. Similarly, the logical sector for track 10, sector 4 is 93. Logical sectors must be specified when using the MS-DOS interrupts 25H and 26H. The interrupt functions 25H and 26H of MS-DOS allow data to be read or written to any portion of the disk.

Absolute Disk Access: Read Operations

Set register AL with the drive number (0 or 1 for A or B, respectively), CX to the number of sectors to read, DX to the beginning logical sector number, and DS:BX to the DTA. Then use INT 25H to read the specified number of sectors into the DTA. All registers are destroyed by this interrupt type (except the segment registers).

You must use the POPF instruction when the function returns control to your program. The status flags are saved on the stack by the interrupt and are not restored prior to returning control. If the carry is set after execution of INT 25H, AL contains the DOS error code. The possible error codes are summarized in Table 8-8 and are the same for the absolute write interrupt (26H).

Absolute Disk Access: Write Operations

Use interrupt 26H to write the information in the DTA to disk. The number of sectors to write are specified in CX, AL contains the drive number, and DX must contain the beginning relative sector for the write operation. The error codes returned, should the operation fail, are the same as for INT 25H. Table 8-8 describes these error codes in detail.

Table 8-8
Error Codes for Absolute Disk I/O
Functions 25H & 26H

Error Code Found in AL	Meaning
00H	Attempted write to a write-protected disk.
01H	Non-existent parameter
02H	Drive not ready
03H	Non-existent command
04H	Data error
05H	Bad structure length
06H	Seek Error
07H	Unknown Media Type
08H	Sector not found
09H	Printer Out of Paper
0AH	Write Fault
0BH	Read Fault
0CH	General Failure

Programming Examples

Here are four routines that use many of the disk function calls just discussed. New system calls that are not disk related are also discussed and demonstrated. The programs have been written to demonstrate how utilities are created.

Have you ever wondered what actually takes place when you type DIR or COPY or some other command from MS-DOS? You have commanded the system to load and execute a utility program. The program performs a specific function: DIR displays the directory, COPY copies the contents of a file or files to another file or files. I'll demonstrate how easy it is to write simple utilities to perform such functions.

The four programs listed in Listings 8-1 through 8-4 found in Appendix D are:

1. NEW_TYPE.ASM, Listing 8-1, a program similar to the MS-DOS TYPE command. It prompts the user for a file specification, opens the text file specified, and displays its contents on the screen.
2. NEW_COPY.ASM, a program similar to the MS-DOS copy command (Listing 8-2).

3. FAST_COP.ASM (see Listing 8-3), a program that copies the contents of one file to another faster than the MS-DOS copy command performs a file copy.
4. DIRREAD.ASM in Listing 8-4, a program that reads the filenames from the directory, sorts the entries in alphabetical order, and displays the names.

Each program demonstrates different facets about disk I/O and user interface techniques, as well as introducing you to a few handy BIOS routines. At times the BIOS routines are a better alternative than are the MS-DOS routines to accomplish a function. Screen graphics for many display functions are better handled by BIOS than by MS-DOS. The same can be said of the limited communications functions MS-DOS supplies.

Programming Example 1:

NEW_TYPE.ASM

The program in Listing 8-1, simulates the MS-DOS TYPE command used in displaying a text file's contents. The program uses the sequential file access.

The Data Segment

The data segment contains keyboard and disk buffer areas. It also allocates space for the file control block required to access the file we want to display. Five messages are also defined and provide the output portion of the program's user interface.

Keyboard Buffer

The keyboard buffer is defined in the usual manner required by the line input function 0AH. MAX_CHARS is to be initialized within the program before invoking the line input function. It signifies the total number of characters to accept from the keyboard.

CHARS_TYPED is the second byte of the buffer area and will contain the actual number of characters typed by the user when the line input function of MS-DOS returns control to your program. K_BUFF is 20 bytes in length and is initialized with ASCII 20H, a space character. Characters entered from the keyboard are placed here.

The Disk Buffer

DISK_BUFFER is the buffer where data read from disk is placed. The program must set the disk transfer address to the start of this buffer before attempting to use the disk I/O functions of MS-DOS. The buffer is 512 bytes in length; this is also the number of bytes in a disk sector. The DUP statement in the data definition initializes all 512 bytes with zero.

Notice that the buffer is terminated with a $. The reason the first byte following the buffer is the dollar sign character (24H) is not obvious yet. I used the MS-DOS line display function (09H) to display the buffer's contents. The function requires a dollar sign be used as a terminator to the string that is displayed. Using this method, you can read a sector from disk and call the line display function to see the contents of the disk buffer. You do not have to move the data from the buffer to another buffer dedicated to video output.

The FCB

The next item defined in the data segment is the file control block. The FCB is defined as a normal FCB (37 bytes in length). The labels used to describe each field in the FCB are descriptive, as shown in the listing. All fields are initialized to zero. As the program operates on only one file at a time, we need to define only one FCB.

Messages

All messages that are defined in the data segment are terminated with the $, as required by the MS-DOS line display function.

The Stack Segment

The stack segment reserves 64 words of stack area for the program. We will not be doing a lot of pushing, nor will we be using the stack to pass parameters. Therefore, only 64 words are reserved, more than enough for the program.

The Code Segment

Now look at Listing 8-1. The program starts in the usual manner at label START_1 by saving the return address on the stack and initializing the segment registers DS and ES. Both are set to point to the base of the data segment.

The BIOS Scroll Function

One of the handiest BIOS routines is the function used in scrolling a window or portion of the video display. The video display is 25 rows by 80 columns wide. The coordinates used by this function are therefore in the range of 0 to 24 for the row designations and 0 to 79 for the columns. Row 0 and column 0 define the upper left-hand corner of the display.

The macro definition for the scroll function is invoked at the label START_2 in our program. The macro definition may look intimidating on its own, but a full explanation of the function is given in the MACFLE.MAC listing and below.

Basically, the function works as follows: Register AH must contain 06H if you want to scroll the display up, and 07H if we want to scroll the display in a down. Next, set AL to the number of lines you want to scroll. If AL is set to zero, the entire window is blanked.

Set register CH to the row of the upper left corner of the scroll window, CL to the upper left column, DH to the row of the lower right corner of the window, and DL to the lower right column number. Register BH must contain the attribute or color byte. The attribute and color bytes are illustrated in the MACFLE.MAC listing under BIOS video routines.

After all the registers have been initialized, execute the instruction INT 10H. It is shown in Listing 8-1 as INT BIOS, which is really a misnomer, as BIOS has several other interrupt types other than 10H assigned to its functions. However, with my infinite desire to use the EQU directive, I assigned the label BIOS to the interrupt 10H in the MACFLE.MAC file.

After the function is executed in the program at label START_2, a message is displayed asking the user to enter the drive and filename of the file to be displayed. Then a carriage return/linefeed is executed to advance the cursor to the next line and the MAX_CHARS is set to 15 prior to issuing the MS-DOS function to accept a line of keyboard input.

Program Label: GET_NAME

The program is now ready to receive the filename from the user. The macro @KBDLINE is executed with the parameter of KEYBOARD, which defines the buffer in the data segment that is being passed to the macro. You can enter a maximum of 15 characters in the standard MS-DOS format for a file specification as follows: d:filename.ext. 15 characters were specified, as the function expects the last character accepted from the keyboard to be a carriage return. Therefore, 15 characters is the maximum that can be typed from the keyboard. The filename given may be less than 15 characters, but it cannot exceed this number.

The PARSE_STRING Function

Now for a little trick in getting the filename from the keyboard buffer, K_BUFF, to the FCB, where it must be placed before we use the MS-DOS function to open the file.

The PARSE_STRING function (29H) parses a string for a filename and places the string in an FCB of your designation. The FCB must not belong to a file that is already open. The PARSE_STRING function requires that the effective address (EA) of the string to be parsed be in the DS:SI registers and the EA of the FCB be placed in the ES:DI registers.

Register AL must contain an encoded 4-bit combination informing MS-DOS as to how the parsing will be carried out. The options available to you are summarized in the MACFLE.MAC. I used the encoding as follows:

```
Bit 0 = 1 = ignores any file separators encountered,

Bit 1 = 1 = leaves the default drive unaltered should the string not
            contain a drive specification,

Bit 2 = 0 = leaves the file name in the FCB unaltered should the string
            not contain a file name,

Bit 3 = 0 = leaves the FCB file extension unaltered should the user not
            specify an extension.
```

You can specify your own method of scanning when using this function by setting or clearing these four bits. You should set the high order nibble to zero.

After setting the register in a manner prescribed, execute the INT 29H instruction. The macro @PARSE_STRING sets the registers for you when you specify the EA of the string to parse, the FCB, and the encoded bit patterns as parameters.

Procedure Call: OPEN_FILE

This procedure invokes the MS-DOS function to open the file specified in the FCB. The filename has been placed in the FCB by the PARSE_STRING function. When expanded by the assembler, the macro @OPEN points DX to the FCB specified as the macro parameter, moves the MS-DOS function code to open a file (0FH) into the AH register, and executes the INT 21H instruction.

After MS-DOS returns control to the program, the AL register is checked for a value of zero. If AL contains zero, the file was found in the directory and opened. If AL contains 0FFH, then the entry was not found in the directory. I used this information to either reset the carry if the entry was found or to set the carry as an indicator to inform the main program that the file was not in the directory.

If the file is not found, a message is displayed stating such, the keyboard buffer is cleared, and the FCB is closed. The carry is set, and control returns to the main program. If the file is found, the carry is reset, and the procedure returns control to the main program.

The fourth source statement in Listing 8-1 following the GET_NAME label uses the carry bit as set by the OPEN FILE procedure to ask you for another filename. The program branches back to the label START_ 2 if the carry was set on returning from the OPEN FILE procedure.

Program Label: SET_UP

Once a valid filename has been entered and the file opened, the disk transfer address is set via the @SET_DTA macro. The parameter supplied to the macro is the buffer for disk I/O, as defined in the data segment, DISK_BUFFER.

Now you must set the current block and record number to the beginning of the file. The value zero is stored in the RECORD_NUMBER and BLOCK_NUMBER fields of the FCB (see the data segment). The RECORD_SIZE field of the FCB is set to 512. When the MS-DOS function to read a record is invoked, an entire sector is read to the disk transfer address. When MS-DOS opens a file, the record size is initialized to 128 bytes (80H).

Program Label: READ_RECORD

The macro @READ_SRECORD invokes the MS-DOS function to read the record specified in the RECORD_NUMBER and BLOCK_NUMBER fields of the FCB into the FCB. Since the record size has been set to 512 bytes, an entire sector is read into the DTA.

On returning from the function, register AL is checked for a value of zero, indicating that the read operation was successful. A nonzero value indicates that an error has occurred. Control is transferred to the code at the label READ_ERROR when an error is encountered or to CONTINUE if there were no errors.

Program Label: CONTINUE

At the program label CONTINUE, the macro @VDLINE invokes the MS-DOS line display function (09H). This is a unique manner in which to use the function. The function is primarily used in displaying messages on the screen. Here we use the function to display the entire disk buffer at one time. That is why the disk buffer is terminated with 24H ($). You can use this function to treat the disk buffer as if it were simply an ASCII message you want to display. Can you see a potential drawback to this technique? (Hint: Don't use the $ character in a file you want displayed using this program. Try rewriting the portion of the program that displays the disk buffer, and fix this potential bug.)

After displaying the disk buffer, the program performs a jump backward to the program label READ_RECORD, and the process is repeated.

Program Label: READ_ERROR

We've followed the program logic flow through a no-error condition after the read operation. Now let's look at the logic flow when register AL contains a nonzero value after a read operation.

We can determine the value in AL by using one of my favorite methods, decrementing the register and checking for zero. The decrement instruction for 8-bit values requires 3 clocks to execute, compared to 4 clocks for a register immediate compare, such as CMP AL,01H. Execution time is not a critical issue in this program, yet I want to bring little programming quickies like this to your attention. Here the execution time for the DEC AL instruction is exactly half that of an instruction that accomplishes the same function—setting the zero flag.

AL can contain one of three possible error codes on returning from the function. If a read operation was attempted and the first byte encountered during the read was an end of file (EOF) character (1AH), then AL will contain 01H and there will be no data in the DTA. If AL contains 02H, there was not enough room in the DTA to transfer one record. This can occur when the DTA is smaller than the record size in the DTA. If AL contains 03H, then a partial record was read into the DTA before the EOF was encountered.

When the EOF is encountered (AL =01H), the program branches to the label EOF_FOUND. A message is displayed announcing that the sequential read operation has been completed, the file is closed, and the program terminates. All files must be closed using the MS-DOS function 10H, which closes the specified FCB and updates the disk's directory. Should you fail to use the CLOSE function, the diskette will not be updated correctly and some or all of your data may be lost. So be sure to close any open files before you terminate a function.

The DTA too small (AL = 02H) error code should not occur in this program, since the DTA is as large as the record size. However, if the error should occur, the program branches to the label DISK_ERROR, which displays a fatal error message before exiting the program.

Partial record read: (AL = 03H) when files are not exactly divisible by the record size, (512 bytes in our program). In this case, the last data in the file does not form a complete record. Therefore, this error code informs the program that it was unable to transfer the requested record size to the DTA prior to encountering the EOF character.

The program branches to the program label DSP_PARTIAL, which displays a partial record before exiting the program. The DI register is used as an index into the disk buffer to point to the next character to be processed. DI is initially set to zero to access the first byte in the buffer. The value pointed to by the base address of the disk buffer and DI is moved into the DL register. DL is compared to 1AH, which is the end of file character used by MS-DOS.

If the character is found, the program branches to EOF_FOUND. If the character in DL is not 1AH, then the program displays the character using the macro @CHARDSP. DI is incremented to point to the next byte in the disk buffer, and the program branches back to DSP_PARTIAL, which fetches the next byte from the buffer. Notice how the indexed mode of addressing is used to access the data in the buffer.

I will not go into as much depth with the rest of the programming examples. However, any new MS-DOS or BIOS functions introduced are discussed in detail for better understanding.

Programming Example 2: NEW_COPY.ASM

This program copies the contents of one file to another and can be found in Listing 8-2. The program is slow. It is useful only for files less than 4K in length. NEW_COPY.ASM demonstrates the sequential read and sequential write functions found in MS-DOS 1.x and 2.0.

One note of caution: because the program illustrates how a copy command is defined, it should not be used to copy large files. The program works, but at a snail's pace. Try the program out on smaller files (4K or less), and you won't have to go get a cup of coffee while the program executes the copy.

In the previous program (NEW_TYPE.ASM), only one FCB was required, as there was only one open file in the program, the file being read from. To perform a copy, we must specify two FCBs in the data segment, as there will be two files open at a time: the source file being read from and the destination file being written to.

I used the same format and descriptors to define the first FCB as in the previous example, except that I refer to the first FCB with the symbolic name FCB_1. The second FCB is naturally referred to by the name FCB_2. Notice in Listing 8-2 that the second FCB uses a D suffix after the field definitions within the FCB. For example, the destination drive number field is named DRIVE_NUMBER_D. The destination fields were named in this manner to differentiate them from the source labels of similar names.

Messages

The messages necessary to prompt the user for the source and destination files are also defined in the data segment. The program even tells the user whether the destination file is new or if it already exits. If it already exists, the file is overwritten, and the previous contents are lost. A more well-mannered copy routine would ask you if you really want to overwrite the file. This is not done in this program, but it would be a worthwhile exercise for you.

Another RAM location, named BYTE_COUNT, is also defined in the FCB. The program uses this location as a record counter when writing to disk. The program initializes the record size field to one in both the source and destination FCBs. Therefore, one record is 1 byte in length. The BYTE_COUNT informs the program when 256 bytes have been written to disk.

Program Label: START_2

In Listing 8-2, the second instruction following the label START_2 invokes a macro, which sets the cursor to the specified screen location. To invoke the macro, supply it with the screen number and the row and column coordinates. A row value of 0–24 or column number of 0–79 is acceptable.

The macro is expanded, and the screen number passed to the macro is placed in BH, the row number in DH, and the column number in DL. The contents of register AH are set to the value 02H and an interrupt 10H is executed. The cursor is set to the position specified on the screen number that was specified.

The program statements appearing at the label GET_NAME prompt the user for the source filename, parse the string, and place the filename entered into FCB_1. If the procedure OPEN_FILE finds the file in the directory, it is opened and the carry flag is reset. If the procedure is unable to locate the file in the directory, the carry is set to indicate an error has occurred. The program displays a message to inform you that an invalid filename was entered and you must reenter the filespec. The program branches to the label START_2, which restarts the program.

Assuming the filename was good, the program continues at the label SET_UP. The disk transfer address is set to point to the DISK_BUFFER, which is defined in the data segment. Only 1 byte is allocated to the DISK_BUFFER. The program works with source and destination record sizes of only 1 byte. Therefore, any record read or write operation will read or write only 1 byte from or to disk.

You might wonder why we do not need to define a second buffer for the destination file. The reason is simple. Since we want to write the data read from disk, we can use the same buffer for reading as we do for writing. When a byte of data is read from the source file, it is written to the destination file. Other applications will no doubt require separate buffers for each file manipulated by your program.

Next, the program invokes the macro to set the disk transfer address. The FCB fields are initialized to the starting block and record number (zero), and the record size is set to one. The program calls the GET_SECOND procedure, prompting the user for the second, or destination, drive. The procedure also creates the file if it does not exist, or, if the filename specified belongs to an existing file, the procedure sets the file's size to zero, erasing the previous contents of the file.

The fields RECORD_NUMBER_D, BLOCK_NUMBER_D, and RECORD_SIZE_D of FCB_2 are initialized to zero, zero, and one. The fields are set to the same values as in the source FCB_1, because we want to start from the beginning of each file when reading from the source file and writing to the destination file. The block and record numbers in each are therefore set to zero. The record size is set to 1 in both the source and destination FCBs to insure that what we read is what we write.

Program Label: READ_RECORD

We are now ready to read a record from disk. This portion of code reads a record from disk and checks AL for error codes. If AL contains 00H, then there were no errors and the program continues. The record just read is then written to disk.

Program Label: CONTINUE

The macro @WRITE_SRECORD writes a record to disk from the DTA. The parameter of the FCB belonging to the file to be written to is passed to the macro. If there are no errors after performing the write operation, the RAM location BYTE_COUNT is incremented.

When the BYTE_COUNT equals 0FFH and is incremented, the counter is said to roll over to zero. When this happens, the zero flag is set in the flag register. The program then executes the instructions to display an asterisk (*) on the screen. This lets the user know every time 256 bytes of data have been written to disk.

After the write operation is completed, the program branches back to the label READ_RECORD, where the next record is read from disk. The entire read/write portion of the program is then repeated.

The program checks for read errors at the label READ_ERROR and write errors at the label WRITE_ERROR. Examine these portions of code and see how I handled errors in the program. Notice in the READ_ERROR portion of code that if AL returns with 02H or 03H after a read operation, the program jumps to the DISK_ERROR routine. The 02H error code signifies that the DTA was too small to hold one record as read from disk. Since we defined a record size of 1, and the DTA size is 1, this error code does not make much sense in this application. Therefore, if it should occur, it is flagged as a hard, or unrecoverable, error, and the files are closed.

Similarly, if the error code is 03H, MS-DOS is informing you that an EOF was encountered and that a partial record is in the DTA. Again, with a record length of 1 having been specified, this error makes little sense. Should it occur during this program's execution, it is flagged as a hard error. If our record size were greater than 1, these errors would have some relevance.

When the EOF character is encountered in the source file, a dash (-) is displayed on the screen, and the source and destination FCBs are closed. The close operation updates the destination file's directory entry properly before the program terminates.

Assemble this program, and try it out on a few small files. Then try it on a large file. Try NEW_COPY.ASM to copy a file of at least 50K in length. Get up, go out, cut the lawn, fix and eat dinner, and then come back and see if the program is finished. The copy does not take quite that long as I just described, but it seems to when you're watching the asterisk appear on the screen every time 256 bytes are written to disk. Surely there must be a faster method to simulate the MS-DOS COPY command. There is and the program is listed in Listing 8-3.

Programming Example 3: FAST_COP.ASM

FAST_COP.ASM uses the random block read and write functions to copy the contents of one file to another. A little trick speeds up the copy process. The record size specified is 1 byte as in the previous example, but rather than reading one 1-byte record at a time, the program reads a block of 1-byte records into the DTA.

Each time the program reads from the source file, the block read function attempts to read 32,767 records into the DTA. Since we are working with 1-byte records, the disk buffer size is defined in the data segment as 7FFFH, or 32,767 bytes in decimal. A key point to remember is that the disk buffer must be set to a size equal to or greater than the product of the record size and the number of blocks read.

If you recall from the discussion of the random block read function earlier in this chapter, the number of blocks to be read is placed in the CX register prior to invoking the function. Be sure that the number of records is nonzero, or the file size will be altered. Also be sure the disk buffer (as defined by setting the DTA) is large enough to store the number of record blocks specified.

As in the previous program, only one disk buffer is used in this example. Two FCBs are still required: one for the source file and one for the destination file.

Program Labels: SET_UP and SET_UP_D

Macro: **@SET_REL_RECORD**

The macro definition invokes the MS-DOS function to set the random record field of the FCB (bytes 21H–24H) to the relative record, as computed by the current block and current record entries in the FCB. The current block and record numbers are initialized to zero before using this function.

To invoke the function, use the macro @SET_REL_RECORD with the FCB you want altered. The program invokes the function twice, once for the source file and once to set the random record number in the destination FCB. When the macro is expanded, the effective address of the FCB is moved into the DX register, and the function code 24H is moved into AH. Next, the INT 21H instruction is executed.

Program Labels: READ_RECORD and CONTINUE

Macros: **@BLOCK_RREAD and**
@BLOCK_RWRITE

The random block read function is invoked via the macro @BLOCK_RREAD at the program label READ_RECORD in Listing 8-3. To use the macro definition, specify the FCB of the file you want to read from, the number of record blocks you want to read (7FFFH, in this program), and the record size. Although the record size field in the FCB is defined after opening the source file, this macro when expanded insures that the record size field is set as desired before the block read is performed.

The macro expansion moves the effective address of FCB_1 into the DX register, the block count into the CX register, and the block read function code 27H into AH. The INT 21H instruction is then executed. If there was an error, the program branches to the label READ_ERROR. The only error that is unrecoverable is when AL contains 02H on return. If AL equals 01H or 03H, then a partial record was read, and CX contains the number of records read.

If the read operation is successful (no errors: AL = 00H), then the program writes the entire block of records to the destination file. The macro @BLOCK_RWRITE is invoked by specifying FCB_2, the number of blocks to write (7FFFH), and the record size (01H) in the argument field of the macro. The only error of interest occurs when there is insufficient disk space available to write the number of records specified.

If a partial record was read because the read operation encountered the EOF character, CX contains the actual number of records read. This value is used by the macro @BLOCK_RWRITE when it is invoked at the program label PARTIAL_3. In this manner, 32K records (bytes) are read every time the program reads from the source file, and 32K bytes are written every time the program writes to the destination file. When a partial read occurs on the last portion of the source file, the number of bytes read will be written to disk, as specified by the contents of CX.

The program terminates in the usual manner by closing the FCBs and returning to DOS. Time this program against the MS-DOS COPY command. I found it to be 2 or 3 seconds faster when copying a large file. A good exercise for you would be to modify the program so that the source and destination filenames can be specified from MS-DOS as part of the command line as follows:

```
A>  FAST_COP B:TESTFILE.FLE TEST2.TXT
```

The filenames will reside in the default FCBs in the PSP of the program. You can then move the names to your data segment and create new FCBs from the names passed on the command line. Remember that when your program begins, the DS register will contain the segment address of the PSP and your program.

Programming Example 4: DIRREAD.ASM

This last program uses the absolute disk access interrupt type 25H to directly read a specified number of sectors from whatever starting sector you specify. The function was discussed earlier in this chapter. Specifically, the program will read the directory of a specified drive, sort the directory entries into alphabetical order, and display the names on the video screen. The source program can be found in Listing 8-4.

Procedure: GET_DRIVE_TYPE

Find the procedure GET_DRIVE_TYPE, near the end of the program in Listing 8-4. This procedure determines the characteristics for the drive number specified by the user. The first byte in the file allocation table describes the drive type (number of sectors per track, single or double density, etc.).

On entry to the procedure GET_DRIVE_TYPE, all registers are saved, and the registers necessary to access the FAT are initialized with the proper values. Register BX must point to the disk buffer where the data read from disk is transferred. AL must contain the drive type, either 0 or 1. CX must contain the total number of sectors to read (one, in our example), while DX specifies the beginning sector number, which is one. Next the INT 25H instruction is executed.

The FAT begins at track 0, sector 1 for all versions of MS-DOS. The FAT always starts on the sector immediately following the boot sector. That is why DX is initialized to 1. Since we are only interested in the first byte of the FAT, only one sector need be read (CX = 1).

The status flags are then popped from the stack, and, if the carry is reset (no carry condition), there were no errors. The data read from disk is in the DTA. At the label NO_ERR2, the first byte transferred to the DTA is transferred to AL. The different possible drive types and the corresponding FAT entries were summarized in the discussion of the FAT earlier in this chapter. The program determines the drive type from the value of this byte.

The data segment locations DIR_SEC, DIR_BEG, and ENTRIES are initialized to the proper values based on the drive type. DIR_SEC contains the total number of directory sectors possible for a given drive type, while DIR_BEG contains the beginning sector number of the directory. ENTRIES is initialized to the total possible directory entries for the drive type, as summarized in Table 8-4.

After the drive type is determined, the program restores the registers and returns to the main program following the CALL instruction to the procedure (find the label ALL_RIGHT in the source file). The absolute disk read interrupt is invoked using the parameters recovered from the first byte of the FAT. DX is set to the value of DIR_SEC, which specifies the number of sectors to be read, and CX is set to the beginning directory sector, DIR_BEG. BX points to the disk buffer, and the the INT 25H instruction is executed. The entire directory is read into the disk buffer. Assuming there were no errors, the program calls the routine to sort the directory entries.

Procedure: DIR_SORT

The procedure DIR_SORT sorts the entries in the disk buffer in alphabetical order. The format for the directory entries is important as the filename is specified in bytes 0–10. If byte 0 of the directory contains 00H, then the entire entry was never used. If byte 0 contains E5H, the entry has been previously deleted.

The procedure skips empty or deleted entries. When an active entry is found, the filename in bytes 1–10 of the directory entry moved to another buffer (DIR_BUF2), which is where the filenames will be alphabetically sorted.

At the program label MOVE_DIR, DI and SI are set to the effective addresses of the destination buffer (DIR_BUF2) and the source buffer (DIR_BUFF). The direction flag is cleared by the CLD instruction. This is in preparation for the MOVSB command, which moves a byte from the DS:SI pointer to the ES:DI pointer. A count value of 11 is moved into the CX register. Eleven is the total number of characters found in the filename field of a directory entry.

Next, REP MOVSB, a repeated move instruction, is executed. Each time the instruction is repeated, a byte of data is transferred from the address formed by DS:SI to ES:DI. The pointers SI and DI are automatically incremented after each MOVSB is executed, and register CX is decremented by one after each MOVSB. The move is repeated as long as CX contains a nonzero value. This is a very efficient method of moving a string from one place to another.

Next, the pointers are adjusted to point to the next directory entry in DIR_BUFF, and the entire process is repeated. After all of the filenames belonging to active directory entries have been moved to DIR_BUF2, the procedure begins the sorting process.

Program Label: DIR_SORT2

Many methods have been devised to sort data efficiently. Perhaps the most popular algorithm is the bubble sort. In effect, two adjacent entries in a list are compared. If the second item in the list is less than the first, the entries are swapped. A flag is set to indicate an exchange took place on this pass through the list.

Next, the second and third entries are compared. If the third is less then the second, the entries are swapped and the flag set. This process continues for all items in the list. After the last two items have been compared, the swap flag is checked. If the flag is set, the list is again compared from the top down. When a pass of the list results in no entries being exchanged, the list is said to be in order, and the sort is complete.

Naturally, if a list can also be sorted in descending order, it can be sorted in ascending order. The option is left to the programmer and dictated by the application.

The portion of code that is of interest in the sort routine begins at the label DIR_SORT_3. The source string is compared to the destination string byte for byte. As long as the characters compared are equal, the operation proceeds until all 8 characters in the filename have been compared. This is accomplished by, REPE CMPSB.

When the procedure DIR_SORT2 is first entered, the pointers to the source and destination strings are pushed on the stack along with the number of directory entries the procedure is to sort. This is significant, as the procedure that swaps the entries acesses these parameters from the stack. I used this method of parameter passing to illustrate paramater passing via the stack.

If the destination string is lower in value than the source string, the procedure calls SWAP_ENTRY to swap the entries. SWAP_ENTRY begins by saving the BP register on the stack and moving the current value of the stack pointer to the BP register. This allows us to access the stack using BP and a displacement value.

Next, the source and destination pointer values are moved from the stack to the SI and DI registers. Now for a little magic. The byte pointed to by DI is moved into AL. AL and the contents of the memory location pointed to by SI are exchanged (XCHG AL,[SI]). Now that AL contains the byte found in the source string, it is saved at the same byte location in the destination string. It's a quick and efficient way to exchange memory operands.

All 11 bytes in the source and destination filename are exchanged in this manner. The exchange flag is set to a nonzero value to indicate that an exchange took place. Study this procedure and the stack parameter-passing technique. You'll find it handy in many programs you write.

After the directory entries have been sorted, the procedure returns to the main program. You can modify this routine to fit almost any application where a bubble sort is required.

Procedure: CHECK_BOUNDS

In Chapter 7, I promised you a procedure that can be used to check bounds on a number. This is the procedure. As illustrated in Listing 8-4, the routine is very simple. It alters no registers except the flag register where the carry bit reflects the result of the comparisons.

The procedure expects to find a minimum byte value stored in RAM location MIN, with the maximum value in the RAM location MAX. The procedure compares the contents of AL against the values stored in these locations. If the value in AL is less than MIN or greater than MAX, the carry is set to reflect that the number in AL is not in the specified range. If AL is in range, the carry is reset when the procedure returns to the calling program. In this program, the procedure is used to compare AL to valid drive numbers as entered by the user.

Procedure: DIR-DSP

This procedure displays the sorted directory entries stored in DIR_BUF2 in the data segment. The routine formats the display so that 4 filenames are displayed on each line. As there are 24 lines available for display on the IBM PC video screen, the total number of entries that can be displayed at a time before the screen begins to scroll is 4 × 24, or 96, entries.

The maximum number of directory entries on a 9-sector, double sided disk is 112. The program pauses after displaying 96 entries and waits for you to press a key before displaying the remaining entries. This avoids unwanted directory scroll. If a directory contains more than 96 entries, the pause gives you a chance to read the filenames before they are scrolled off the screen.

The program's documentation, as found in Listing 8-4 explains each program line in detail. You will now be able to experiment with the functions MS-DOS provides in handling disk I/O in your own programs. Use these examples as the basis for future experimentation. In the next chapter, I'll discuss the BIOS functions as used in the example programs presented. These functions and others can be found in the macro file of Listing 6-1.

Chapter 8 Review

1. Using MS-DOS 1.0 and 1.1, a disk ___________ ___________ must be established in memory before attempting a disk read or write operation.
2. The purpose of the DTA is to provide a ___________ for data transfers to ___________.
3. Name three types of disk access. Briefly describe each. (Compare your answer to the description given in the text.)
4. Using MS-DOS 2.0 you still need to define a DTA. (True or False)
5. Using MS-DOS 2.0 you still must define an FCB. (True or False)
6. You can find the macros for disk functions listed in Chapter ___________.
7. A utility is actually a mini-___________.

9

The Basic Input/Output System

In this chapter I want to deviate from the functions used under MS-DOS. While it provides a very complete set of user functions for disk I/O and character oriented keyboard and display processing, MS-DOS lacks routines that efficiently allow you to use graphics and sound. The BIOS functions used in this chapter can be found in MACFLE.MAC (Listing 6-1), following the MS-DOS functions.

The BIOS functions are one level above the actual hardware implementation of the machine. As mentioned in previous chapters, MS-DOS functions are the highest level (from the AL programmer's standpoint), with BIOS being the next level. MS-DOS uses the BIOS routines to perform the functions discussed thus far, with the exception of disk I/O, which is handled entirely by MS-DOS.

This linkage to BIOS provides you with a layer of isolation from the operating system. BIOS communicates directly with the devices attached to the system. MS-DOS communicates with the system resources through the Basic Input/Output System. Portions of BIOS are implemented differently from machine to machine. Therefore, many programs using BIOS will run on only the machine for which the program was written. Other 8088/86 or 80186/MS-DOS machines may not be able to run the program, due to a different implementation for a given function.

A good example of low level incompatibility is the CONFIGSY.ASM program discussed in Chapter 2 (see Listing 2-1, Appendix D). The program uses BIOS interrupt 11H to return a binary word in AX that describes the system's resources. Each bit or combination of bits in the word is set or reset depending upon whether a particular resource is available to the system. The significance of each bit in the word is different on the TANDY 2000 MS-DOS machine and the IBM PC. If the program in Chapter 2 were assembled and run on the TANDY 2000, the information returned by the function would be different than for the IBM PC. Table 9-1 summarizes the bit significance of each bit returned by this function on the two different machines.

Table 9-1
Equipment Flag Differences
BIOS Interrupt 11H

Bit #	**Tandy 2000 Meaning**	**Bit #**	**IBM PC Meaning**
0	TV/Joystick	0	Disk Drives Present
1	Monochrome Graphics	1	Not Used
2	Monochrome Graphics with Color Option	2 & 3	Planar Ram Size 00=16K, 01=32K
3	Floppy Drive 1		10=48K, 11=64K
4	Floppy Drive 2	4 & 5	Video Mode
5, 6, 7	Not Used		00=Unused
			01=40 × 25 B/W, Color Card
			10=80 × 25 B/W, Color Card
			11=80 × 25 B/W, Monochrome
		6 & 7	Number of disk drives.
8	B/W Monitor	8	Unused
9	Color Monitor	9, 10, 11	Number of RS-232 Cards
10	TV Monitor		
11	Not Used		
12	Joysticks	12	Joysticks
13	Printer	13	Not Used
14	Not Used	14 & 15	Number of Printers
15	Not Used		

Table 9-1 shows the differences between the Tandy 2000 and the IBM PC in the interpretation of the equipment flag returned by BIOS INT 11H. Both machines are MS-DOS compatible, although the BIOS functions are implemented differently.

BIOS must be used to perform those functions MS-DOS lacks—graphics, communications, etc. While MS-DOS provides functions for sending a character to the printer port (function 05H, INT 21H) and to the communications channel (INT 21H, function 04H), the functions MS-DOS supplies are minimal and not suitable for a wide variety of programs.

For example, the serial I/O routines link themselves to BIOS and look for certain voltage levels to be present at the RS-232 connector of the serial board. These signals are named Clear To Send (CTS), and Data Set Ready (DSR). They are usually supplied by a modem (Data Circuit Equipment, or DCE) attached to the PC. If you want to transmit a character via the serial board and DSR and CTS are not present, the BIOS transmit routine waits a specified amount of time for the signals to be asserted. If after the specified time, the signals are not sent, the routine returns, and the character is never sent.

Similarly, when you want to receive a character from the communications channel, the MS-DOS function 03H could be used. Other processing chores are temporarily suspended while the routine waits for a character to arrive at the serial port. This delay could prove to be critical if other processing chores must be executed in real time (without delay).

The delay is unacceptable for another reason. Since characters arrive in an asynchronous manner, their arrival is not predictable. Therefore, your program must continually sample the communications port via the MS-DOS 03H function, so as not to miss an arriving character. Given the delay inherent in the function when a character is not ready, your program would do little else besides polling the serial port for a character.

Since the serial I/O functions link themselves with BIOS routines, the BIOS routines are responsible for the manner in which the call executes. For serial I/O, it is best to skip the MS-DOS and BIOS routines altogether and directly access the serial communications board and the 8250 UART (Universal Asynchronous Receiver Transmitter), an integrated circuit responsible for handling serial communications. I know this sounds contradictory to my previous argument of using MS-DOS function calls to maintain compatibility between MS-DOS machines; however, certain MS-DOS functions simply do not provide you with the power you need to perform a function from within the program. Serial communications is the subject of the next chapter, where I'll show you how to install your own RS-232C interrupt handler and overcome the communications handicaps MS-DOS and BIOS impose upon you.

Other functions, such as scrolling a portion of the display or setting the cursor to a specified position on the display, are totally absent from MS-DOS. You must use the BIOS routines to perform these functions, or directly program the chips that carry out these functions. The further from MS-DOS your program deviates, the more likely that the program will run on only one type of machine. Because the chips themselves may not be at the same port address on every machine, the programs written for an IBM PC may only execute on that machine and not one of the many so called compatibles.

With all these caveats in place, I'll now discuss some of the BIOS routines as they pertain to the video display and the printer. You can use these functions to write graphic routines and spice up your video output. They give you complete control over the monochrome and color adapters used in the IBM.

Printer I/O

There are two printer adapters used in the IBM PC. The first is found on the monochrome display adapter. The printer is mapped to the following port addresses:

Port	Function
03BCH	Printer data - Output
03BDH	Printer status - Input
03BEH	Printer control -Output

When the parallel printer adapter board is installed in the system the port assignments for the printer are as follows:

0378H	Printer data - Output
0379H	Printer control - Output
037AH	Printer status - Input

In most of your programming endeavors, the ports will not be directly accessed. Instead you will use one of the MS-DOS or BIOS routines to perform printer I/O. The only MS-DOS function call for the printer port is function 05H (INT 21H). This function outputs a character in DL to the printer, and it has been incorporated into a macro call in Chapter 6.

If you want to exercise more control over the printer port, you can use BIOS interrupt 17H and one of the three functions BIOS provides. You can send a character in AL to the printer by moving the function code 00H into AH and executing INT 17H. This method is preferred over the MS-DOS function call, because the status of the printer port is returned in AH as follows:

b0 = 1 = Time out

b1 = 1 = Undefined

b2 = 1 = Undefined

b3 = 1 = Select error (printer off-line)

b4 = 1 = Printer selected

b5 = 1 = Paper out condition

b6 = 1 = ACK

b7 = 1 = Printer busy

The port status is returned as above when the read status function is executed (BIOS function 02H, INT 17H). To reset the printer, move 01H into AH and execute INT 17H. An example of a printer I/O is given in the next chapter.

Video I/O

The Monochrome Adapter

Two different video boards are used with the IBM PC. The first is the monochrome adapter and allows the display of text (alphanumeric characters) on a black and white screen formatted as 80 columns by 25 lines. The adapter contains 4K of video memory, starting at the absolute address of 0B0000H.

Two bytes are required to fully describe each character appearing on the screen, and only one 80 by 25 page of text can be held in the adapter's memory at a time. Since 80 × 25 = 2000 characters, and 2000 characters × 2 bytes = 4K, the memory required per screen is 4K.

Monochrome Character Attributes

The adapter stores not only the characters that are displayed but also each character's attribute. The attribute byte tells the display logic whether the character is to be displayed in normal mode (white on black) or reverse video (black on white). The attribute also dictates whether the character is to be underlined or blinking, or both and specifies the intensity level (normal or high) for that character's position. Characters are stored at even addresses in the adapter's memory buffer, and the attribute codes are stored at odd memory locations in the buffer (i.e., 0B0000H = character, 0B0001H = attribute).

The attribute byte is formed as follows:

```
B7 = 1, the character blinks

B6  B5  B4 = Background

1   1   1 = White

0   0   0 = Black

B3 = 1, then the character is displayed in high intensity
```

continued

```
B2  B1  B0 = Foreground

1   1   1  = White

0   0   0  = Black
```

The Color/Graphics Adapter

The color/graphics adapter board allows either alphanumeric text, or graphics to be displayed. You can use a special high resolution RGB monitor, or you can use an RF modulator to drive a color television. The format when displaying text on a television is limited to a 40 character by 25 line display. A television set lacks the resolution necessary to display more than this amount of text on screen. Through the BIOS calls listed in Chapter 6, you can specify the mode of operation. Table 9-2 summarizes those modes as used with the BIOS interrupt 10H.

Table 9-2
Modes of Video Operation
BIOS INT 10H

Function Number: 00H = Set Mode
Function Number: 0FH = Read Mode

Video Mode	Contents of AL
40 × 25 B/W	00H
40 × 25 Color	01H
80 × 25 B/W	02H
80 × 25 Color	03H
320 × 200 B/W Graphics	04H
320 × 200 Color Graphics	05H
640 × 200 B/W Graphics	06H

The color/graphics board contains 16K of memory, which stores alphanumeric video information in a manner similar to the monochrome board. Since the color adapter contains four times the amount of memory of the monochrome adapter, up to four pages of text can be stored in the memory of the color board when using the 80 by 25 display format, or eight pages of text when using the 40 by 25 format, as shown below.

```
*************************************************************
*  80 x 25 = 2000 characters, 2000 characters x 2 bytes = 4K, *
*                                                           *
*  16K/4K = 4 pages, or screens, of text.                   *
*                                                           *
*  40 x 25 = 1000 characters, 1000 characters x 2 bytes = 2K  *
*                                                           *
*  16K/2K = 8 pages, or screens, of text                    *
*************************************************************
```

The BIOS functions presented in MACFLE.MAC and in this chapter demonstrate how to set the active page from the 4 or 8 possible pages in the adapter's memory.

Color Character Attributes

The character attribute associated with each character is summarized in Table 9-3. You can specify any one of 16 colors as the foreground color (character color), and any one of eight possible colors as the background color. As you'll see in a moment, the color attribute byte is slightly different depending on which of the two available graphic modes the adapter is operated in. Should you specify a black background and a white foreground, the attribute byte is similar to the monochrome attribute byte previously discussed.

Table 9-3
Color Attribute Byte

```
              B7   B6   B5   B4   B3   B2   B1   B0
              B    R    G    B    I    R    G    B
              --   -------------- -------------------
Blink -----|            |                   |
                        |                   |
Background Color ----                       |-------Foreground Color
 (Possible 8 Colors)                         (Possible 16 Colors)
```

IRGB - Color
0000 = Black
0001 = Blue
0010 = Green
0011 = Cyan
0100 = Red
0101 = Magenta
0110 = Brown
0111 = White
1000 = Gray
1001 = Light Blue
1010 = Light Green
1011 = Light Cyan
1100 = Light Red
1101 = Light Magenta
1110 = Yellow
1111 = White (High Intensity)

Table 9-3 shows the color attribute byte used when displaying alphanumeric text with the color adapter. Also shown are the color combinations of the 16 possible colors encoded using the IRGB field of the attribute byte.

Graphic Modes

When the color adapter is operated in the graphics mode, the display is divided into a two-dimensional array of pixels (picture elements). The graphics resolution can be set to either medium resolution, providing for 320 by 200 pixels (64,000 graphic points), or high resolution, offering 640 by 200 pixels (128,000 graphic points). The high resolution graphics mode has one drawback. You can display images in black and white only. There is not enough memory on the adapter to describe an attribute color along with 128,000 pixels. This is because 128,000 bits is equivalent to 16K bytes of memory, or all the memory available on the adapter.

Medium Resolution Graphics Mode

Table 9-4 summarizes the color combinations available in the medium resolution graphics mode. Any one of the 16 colors may be specified as the background color, while the foreground color must be from one of the two palettes shown. A total of four different colors can be displayed at any given time using the medium resolution graphics mode of operation.

Table 9-4
Medium Resolution Color Combinations

Background Color = Any of the 16 possible colors defined in Table 9-3.

Foreground Color = Palette 1: Cyan, White, Magenta.
Palette 2: Green, Red, Brown.

Shown are the color combinations available in the medium resolution video mode. To set the color palette, set the registers as follows:

AH = Function number 0BH.
Register BH = 00H
Register BL = 00H = Palette 2
01H = Palette 1

and execute BIOS interrupt 10H.

High Resolution Graphics Mode

The high resolution mode of operation allows for a graphics display of 620 columns by 200 rows of pixels. The main advantage to this mode is that the resolution is twice that of medium resolution. The disadvantage is that the high resolution screen does not allow the use of color. The only colors available are black and white.

You can output text to the video display in either the high resolution or medium resolution graphic modes. If the characters output to the display are within the range of 00H to 7FH, the characters will be displayed on the graphics screen (see the ASCII chart in Appendix A for the character representation of each code).

Pixel Positioning

When specifying a pixel's position on the screen in either high or medium resolution modes, a row and column coordinate must be used. A medium resolution display consists of 320 columns by 200 rows, and the high resolution display consists of 640 columns by 200 rows. As an example of specifying a pixel's position, examine the statements which make up the macro @SET in Listing 6-1.

The macro's body is comprised of the following instructions:

```
MOV   DX,ROW          ;Get the row number (0-319 med. res.)
                      ;(or 0-639 in high resolution mode)

MOV   CX,COL          ;Get the column number (0-199)

MOV   AL,COLOR        ;Get the color of the dot

MOV   AH,0CH          ;BIOS function code to set a pixel

INT   10H             ;BIOS interrupt.
```

This function colors a pixel at the specified row and column. The function can be used to set or reset a pixel, depending on the color specified. If, for example, the mode was set to medium resolution, and you are working with a red background and specify the pixel's color as red, the dot will disappear into the background color. However, if you specify the color to be brown, the dot will appear in the foreground on a red background.

The color of the pixel when using the high resolution mode must be either black on black (00H), white on black (07H), white on white (77H), or black on white (70H).

Interrupt Type 10H

By now you may feel that programming graphics and text is much too complicated and is better left to BASIC. You would be right and wrong if you felt this way. You'd be right, because the statements provided under BASIC are actually graphic subroutines that allow you to draw lines, circles, and rectangles. They are easy to use and quite efficient. Programming each pixel position on a high resolution screen containing 128,000 pixels is extremely tedious.

This type of programming is repetitive and something that is ideally suited for a computer. If BASIC already contains the routines necessary to compute arcs and perform line drawing functions, why even contemplate writing Assembly Language graphic routines? The answer is speed. BASIC is too slow to perform animation or other graphic chores in a reasonable amount of time.

Therefore, the BIOS functions that allow you to control the video display in both text and graphic modes are what you need to know about when creating high level graphic routines. These functions have been incorporated in the macro calls presented in MACFLE.MAC and are listed in Table 9-5. You should review these functions before continuing.

Table 9-5
BIOS Video Functions
BIOS INT 10H

Function Number	Function
00H	Set Video Mode
01H	Define Cursor
02H	Set Cursor Position
03H	Return Cursor Position
04H	Return Light Pen Position
05H	Set Active Page
06H	Scroll Screen Up
07H	Scroll Screen Down
08H	Return Character Attribute at Cursor
09H	Set Character Attribute at Cursor
0AH	Set Character at Cursor
0BH	Select Palette
0CH	Set/Reset Pixel
0DH	Return Pixel
0EH	Write Character
0FH	Return Video Mode

A Programming Example: GRAPHIC.ASM

GRAPHIC.ASM (see Listing 9-1 in Appendix D) illustrates how higher level graphic routines can be constructed that approach the power and ease of use of graphic statements found in BASIC. Programming graphics in Assembly Language, and is usually a repetitive process. By writing programs or procedures capable of drawing a line, given start and stop values; or a circle, given the radius, aspect ratio, and center point; graphics programming becomes a pleasure rather than a chore.

The program defines a procedure DRAW_LINE, which draws lines on the screen. A macro, ERASE, deletes a graphics line. The procedure and macro both require start and stop values for row (X-axis), and column values (Y-axis). A color must also be specified. The macro ERASE also uses the DRAW_LINE procedure. One procedure can serve to set or reset a range of pixels on the screen.

Since the program uses the high resolution graphics mode, the only colors available are white and black. I used a black background and a white foreground when turning a pixel on, and a black background/foreground combination to turn the pixel off. The allowable values for row and column coordinates are 0–639 for the column, and 0–199 for the row. If the medium resolution mode were used in the program, the column values could range from 0–319, with a row value in the range of 0–199.

Take a look at the program label START. After the segment registers are initialized, the program invokes a macro, @VDMODERD, which uses the BIOS interrupt type 10H and the function number 6H to obtain the current video mode. The value returned in AL defines the current mode as follows:

```
AL = 0 = Text 40 x 25 B/W(EP)

AL = 1 = Text 40 x 25 Color(EP)

AL = 2 = Text 80 x 25 B/W(EP)

AL = 3 = Text 80 x 25 Color(EP)

AL = 4 = Graphics 320 x 200 B/W(EP)

AL = 5 = Graphics 640 x 200 Color(EP)

AL = 6 = Graphics 640 x 200 B/W(EP)
```

The active column number is in AH, and the active page is in BH. Whatever the mode and active column is, we want to save the value. Therefore, after the mode read function is invoked, the value returned in AX is saved on the stack. Since we are working with screen 0, the screen that was in effect, we will not save BX on the stack. Before the program terminates, the previous video mode will be restored.

Another BIOS function to set the video mode is now invoked (function 0, INT 10H). The video mode is set to the high resolution (640 by 200) black and white mode. Using the table just discussed, a value of 6 is moved into the AL register, which informs the function of the mode desired. The remaining portion of the main procedure initializes memory locations STARTX, STARTY, STOPX, STOPY, and COLOR before calling the procedure DRAW_LINE. These RAM locations define the characteristic of the line to be drawn.

The procedure first draws a large A in the middle of the screen (program label DRAW_IT to DRAW_2). Next, a line is drawn from left to right on the bottom row of the screen. The program delays for about 1/2 second, then erases the line. The program again delays for 1/2 second before drawing a new line from the bottom left of the screen to the top left. This process is repeated in such a manner as to give the effect of a moving line creating a marquee around the border of the screen.

This is the simplest form of animation. An object (a line, in this program) is drawn, erased, and redrawn somewhere else on the screen. More complex forms of animation account for more than one object and move more than one object around on the display simultaneously.

After executing the border loop 40 times, the program invokes the macro @WAITKEY, which waits for a key to be pressed. Once a key is pressed, the program reinstates the old video mode of operation that was saved on the stack at the beginning of the program. The program terminates and returns to DOS.

The DRAW_LINE Procedure

This procedure expects that the RAM locations STARTX, STOPX, STARTY, STOPY,and COLOR have been set up prior to calling the procedure. The routine automatically calculates the direction in which the line must be drawn. Since the X-axis coordinates begin with column zero on the far left of the screen and increase in value toward the right of the screen, the direction (with reference to the X-axis) that you want to draw the line is easily calculated from start and stop values for X. If, for example, the starting column value of X is 99 and the stop value for X is 174, the line must be drawn from left to right. If, however, the start value were 174 and the stop value 99, the line would be drawn from right to left. The direction can be calculated by subtracting the start value from the stop value. The direction of the move is from left to right if the result of the subtraction is positive (STOPX − STARTX > 0). Similarly, the direction is from right to left if the result of the subtraction is negative (STOPX − STARTX < 0).

The direction the line is drawn along the Y-axis is calculated in an manner identical to the X-axis. The exception is that the values stored in RAM at STARTY and STOPY are used in the calculation. The row values begin with zero at the top of the screen and increase in value towards the bottom of the screen (see Figure 9-1).

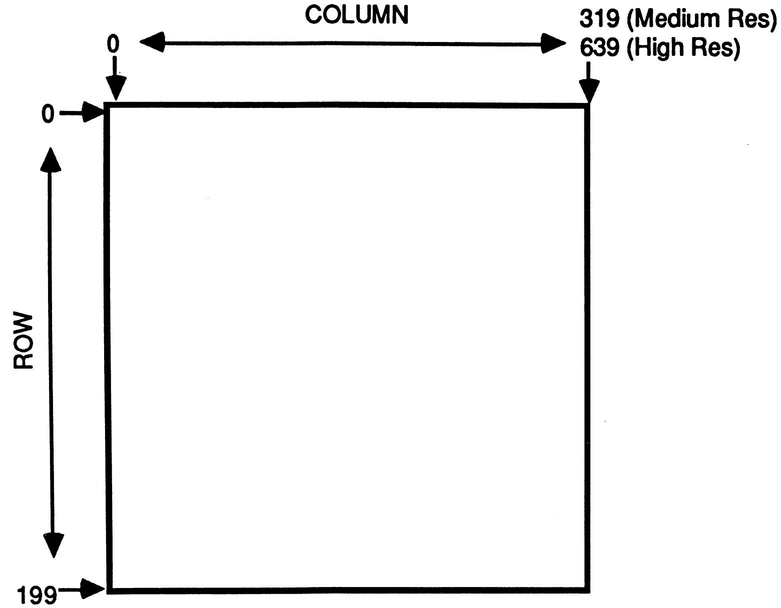


Figure 9-1 Graphic Modes

These calculations are carried out at the beginning of the procedure. The RAM location DIRECTIONX contains a zero value if the X-axis direction is calculated as a move from left to right across the screen (forward) or a nonzero value if the move should be from right to left (backward). The RAM location DIRECTIONY contains a zero value if the Y-axis move is calculated to be from the top of screen to the bottom (down), and a nonzero value if the move is calculated as a move from the bottom of the screen to the top (up).

Once the direction values have been set, the procedure (at label D_L_3) begins to draw the line specified. The macro @SET is invoked to set the pixel at the X/Y coordinates specified and in the color specified. The contents of RAM locations STARTX, STARTY, and COLOR are the parameters used in the argument field of the macro.

A byte in RAM (D_L_FLAG) informs the procedure whether all the values along either the X-or-Y axis have been exhausted. Bit 0 of D_L_FLAG is set when all the X values have been used, and bit 1 is set when all the Y values have been used. If, for example, there are 16 points to be graphed along the X-axis and 20 along the Y-axis, a line similar to that shown in Figure 9-2 will be drawn.

The procedure continues to increment the contents of memory locations STARTX and STARTY until their values equal STOPX and STOPY, respectively. If the initial values of the start positions are larger than the stop positions, STARTX and STARTY are decremented until they equal STOPX and STOPY. The loop from label

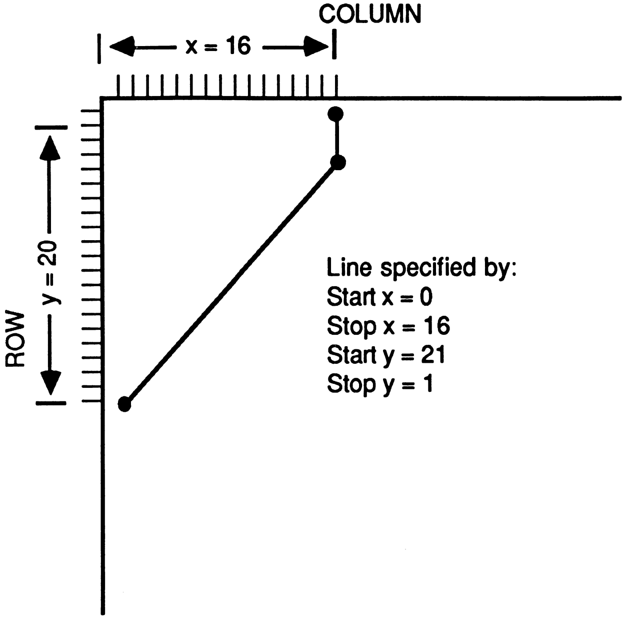


Figure 9-2

D_L_3 to U_2 is executed until the start and stop coordinates are equal. The procedure then returns to the calling program.

The ERASE Macro

This macro erases a line by specifying the start and stop X/Y coordinates and a color combination of black on black. The foreground/background color combination effectively erases the pixel specified by the X and Y values. Notice in the macro definition just prior to the code segment, a call is made to the procedure DRAW_LINE. The same procedure that is used to set a range of pixels can also be used to reset pixels. The only difference between setting and resetting a pixel is in the color code specified. White on black sets the pixel, and black on black resets the pixel.

Going Further

Study the macros used in the program (see also MACFLE.MAC). When you understand how a line is specified and drawn, modify the program as follows: You may have noticed when running this program that the line drawn on the right-hand side of the screen is not flush with the rightmost column (639). Rewrite the portion of the program that draws this line, so that it is flush with the right side of the screen.

This program should give you some ideas of how to automate the many routine and tedious tasks which are involved in creating what are known as graphics primitives. Currently, there are no formal standards for graphics, although there are many proposals under study by the standards organizations. You might want to attempt to write a general purpose procedure to draw circles or some other geometric shapes.

Anyone could have shown you how to set a pixel on the screen through BIOS interrupt 10H/function 0CH. What I wanted to do was to go the next step in Assembly Language graphic programming, graphics primitives.

SOUND

Like graphics, MS-DOS also lacks functions to allow Assembly Language control over the internal speaker of the IBM PC. To produce sound that varies in frequency and tone, the 8255-A Programmable Peripheral Interface (PPI) chip must be directly accessed. This means you can't create programs that can run on other MS-DOS machines, since you are not dealing with MS-DOS when you go directly to the chip to produce sound. The chances that a similar design exists in other MS-DOS machines cannot be guaranteed (although the COMPAQ computer does have a compatible design).

The *IBM Technical Reference Manual* contains a description of the PPI bit assignments and the speaker interface circuit on pages 1-12 and 1-23. You can turn the speaker on and off by gating either the 8253 timer output to the speaker or by setting and resetting bit 1 of port B of the PPI at port 61H. The port can be read to obtain the current setting of the bits for the B port of the chip. This value should be restored after your program is finished generating the sounds at the speaker.

If bit 0 is set, the 8253 timer's output is gated to the speaker. Bit 1 must also be set to enable the gating of the timer's output to the speaker. In the program SOUND.ASM (see Listing 9-2 in Appendix D), the first method is used to generate sound via the internal speaker.

The Data Segment

The sound generating procedure expects a frequency and a duration to be specified in the RAM locations FREQUENCY and DURATION. The values in the table FREQUENCY_TABLE and DURATION_TABLE produce a series of tones of varying duration at the speaker.

The frequency table is one word larger than the duration table. I used 0000H as a terminator word to the table. The program terminates on seeing the end of data value 0000H in the frequency table. Therefore, frequency values of 0000H are not allowed.

The Code Segment

The program starts by clearing the screen and displaying the message "Hit any key to begin SOUND!" The @WAITKEY macro is invoked (see Chapter 6), which waits for a key to be pressed.

After a key is pressed, the program moves one value each from the frequency table and duration table to the RAM locations DURATION and FREQUENCY. Once these locations are set up, the procedure SOUND_GEN is called. This procedure produces the tone at the speaker.

The pointers into the two tables are then adjusted, and the next value in the frequency table is compared for 0000H. If found, the program terminates and returns to MS-DOS. If the value is not 0000H, the program jumps back to the label NEXT_NOTE.

Procedure: SOUND_GEN

Sound generation occurs by setting and resetting bits 0 and 1 of port 61H. A delay is executed after setting the bits, and again after resetting the bits. This delay determines the frequency of the note. The value stored at FREQUENCY determines the amount of time spent in the delay loops at FREQ_OUT and FREQ_OUT_2. The entire process is repeated by a count defined in DURATION.

Interrupts are disabled at the beginning of the procedure and enabled prior to exiting the procedure. If this action were not taken, undesired interrupts could occur while the speaker is either on or off, causing the duration and the frequency of the tone to vary. The result would be an unstable tone, as the delays used in producing the tone would no longer be predictable.

You can incorporate the SOUND_GEN procedure in your own programs. Simply set aside storage in your data segment for FREQUENCY and DURATION. Set these locations with the values desired and then call the procedure.

In the next chapter, I'll discuss the best of all worlds: MS-DOS, BIOS, chip level programming, and interrupts. All the concepts discussed thus far and some new ones will be demonstrated in a versatile data communications program.

Chapter 9 Review

1. BIOS is an acronym for: __________ __________ __________ __________.
2. BIOS insures program compatibility through high level language statements. (True or False)
3. How many pages of text can be stored using the 80 by 25 mode of the color graphics adapter?
4. There are __________ pixels available using the high resolution video mode.
5. The monochrome adapter uses a 320 by 200 medium resolution display. (True or False)

10

Communications

IBM is perhaps best-known in the computer industry for its ability to have its may diverse computing devices communicate with each other. This is not easy to accomplish. This chapter introduces you to the world of data communications and how to write your own communication programs in Assembly Language on the IBM PC.

I stated earlier that it is nearly impossible to program an application if you as the programmer do not possess a working knowledge of the application. I have refrained from including application specific programming examples thus far; however, in this chapter I will concentrate on a full-featured telecommunications program.

There are several other books covering the subject of data and telecommunications more thoroughly than I can in this one chapter. Most notable are those from James Martin, whose books *Telecommunications and the Computer* and *Design Strategy for Distributed Data Processing*, are listed in the bibliography of this book. Martin is a recognized authority on the subject of data communications, and I refer you to his books for further information on the subject.

My intent in the first part of this chapter is to provide you with a background broad enough so that you will understand the terminology associated with data communications. I encourage you to take this background information and the programming example in this chapter and improve upon both.

The second part of the chapter details how the programs COMM.ASM (Listing 10-1) and DLOAD.ASM (Listing 10-2) operate. These two listings can be found in Appendix D. To use the programs, type in the source listings. Notice that certain variables and program entry points are marked PUBLIC and EXTRN in each module. This allows each module to reference variables that are defined in the other.

Once you have created the source files, use the IBM Macro Assembler to assemble the source modules into relocatable (.OBJ) modules. Once the object modules have been created, use the linker to link the files. To link the files type:

```
COMM.OBJ + DLOAD.OBJ
```

at the linker's prompt. Name the executable output file COMM.EXE.

Communications Protocol

In order to communicate effectively, all the parties involved must agree on how the transfer of information is to take place. This set of rules is referred to as a communications protocol. Without it, neither party could effectively or efficiently communicate. I like to think of a communications protocol in terms of two people speaking different languages who are communicating without the ability to see each other. Imagine the problems involved in trying to ask the other person for a glass of lemonade. Without a protocol that has been agreed upon prior to communications, you'd probably wait years to get your lemonade. A protocol must be defined.

You must agree on the language you both will speak. If neither party is bilingual, you still have a problem. For computers, the most commonly used standard is ASCII, or the American National Standard Code for Information Interchange. ASCII is a 7-bit coded character set defining not only characters and number representation but also the use of control functions to facilitate the data transfer. Yet even with computers there may be a problem if they are expected to communicate in an international environment. Like you and your nonbilingual counterpart, they may require some type of code translator or a different protocol.

Code translation is a problem. Although ASCII is one of the most widely used forms of data coding, there are other methods of coding the data for information interchange. Telex uses the 5-bit Baudot code to represent data. IBM developed EBCDIC, which is still another code set. If you were using an ASCII terminal and wanted to communicate on a Telex network, code translation would be required. The 7-bit codes sent from an ASCII terminal to one expecting to receive 5-bit Baudot would fall on deaf ears (or modems), so to speak. Another potential problem exists when the data are exchanged in a character set that was previously agreed upon, such as ASCII, and there are differences in national usage of certain characters within the set.

When do you talk, and when does the other party talk? Or is one party only a listener (receive only, or RO) and never talks? For computers, this correlates to full duplex, half duplex, and simplex operation. Full duplex allows two parties to transmit data to each other simultaneously. Both may talk at the same time, and they are allowed to talk about totally different subjects (remember the last time you asked your boss for a raise?). Half duplex dictates that only one party can transmit at a time. When the first party is finished talking (or transmitting), the other party can begin transmitting data. Simplex is a one-way-only type of communications. Receive-only devices (printers) or send-only devices (keyboards) are typical of devices operated in a simplex mode (Figure 10-1).

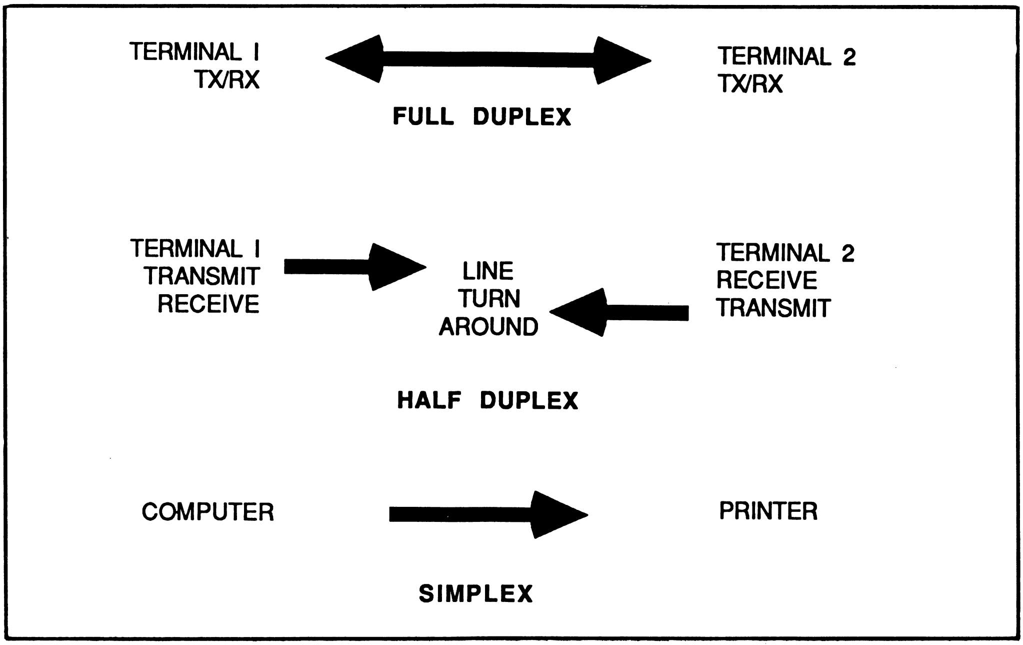


Figure 10-1 Full duplex, half duplex and simplex operation.

There are several other specifics that should be included in the protocol. Will there be error checking? If so, what type? Imagine if you were asking the head of state of a foreign country for a glass of lemonade, and your translator conveyed the wrong information to him and told him he was a real lemon. If the translation contains only a minor error, it may go unnoticed. If it is a big blunder, you could cause an international incident. When your computer transmits data, an error-checking protocol, such as parity checking, checksums, or more advanced methods, may be employed to detect and even correct errors. Data integrity is extremely important in communications of any type. There are many more types of protocols, some of which I'll discuss later.

Methods of Data Transmission

Computers are able to move data in and out of memory or to and from a peripheral device, such as a printer, over multiple lines. This type of structure is referred to as a bus and contains multiple signal-carrying lines that run in parallel (Figure 10-2). Parallel bus structures are an efficient way to move data around inside your computer and to devices that are connected locally to it. An advantage to parallel transmission of data is that whole words, or bytes of information, are transferred at one time at very high speeds.

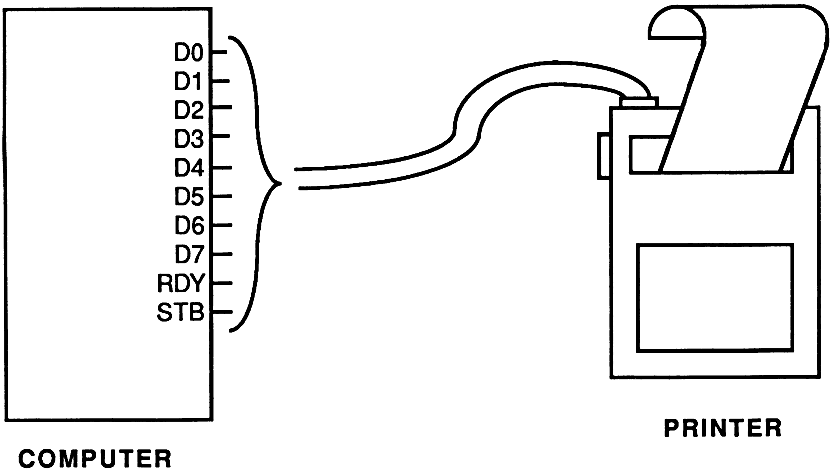


Figure 10-2 Paralell transmission of data from a computer to a printer.

Eight data lines transfer the character to be printed, and the STB signal is used to strobe the signal into the printer's character buffer. When the printer's buffer is full or the printer is busy printing characters, the RDY (ready) line is disabled, informing the computer that the printer is not ready to receive any more characters. When the computer senses the RDY signal to be true, further data transfers will commence.

Each parallel line of the bus corresponds to a single bit of information of the data being transferred. A system can have more than one bus. Data, address, and control bus structures can all coexist as independent structures within the system.

When there is a long distance between the computer and an auxiliary device, parallel transmission of data becomes impractical. Being susceptible to noise, each signal also suffers from transmission line reflections, distortion, and line loss. The physical limitations of the medium (eight or more wires, as opposed to two for serial transmission) and the required in-line amplification to compensate for signal loss present a justification for a different type of transmission over long distances.

It's also more costly to run eight or more lines than it is to run only two wires over a long distance.

Even communicating between the computers I own, demonstrates the advantages of serial versus parallel communication paths. One of my computers is in my downstairs office, while the other is located on the second floor. The physical length of the transmission path is less than 20 feet, yet serial communications gives me the advantage of running only three wires as opposed to eight or more. I only need one wire for the transmit data wire, one for the receive data, and a ground. This is one of the simplest cabling schemes for serial communications, as you will see when we investigate the RS-232C serial interface standard later in this chapter.

In serial communications, each bit of the data word is transmitted in succession, with the LSB (least significant bit) transmitted first. This is done so over one wire (there is another wire for common ground). As shown in Figure 10-3, a character is transmitted in an asynchronous format by first sending what is known as a start bit to the receiving terminal. The start bit enables the receiving terminal to recognize the character bits which follow. The data are transmitted a single bit at a time. In Figure 10-3 the 7-bit ASCII letter N is being transmitted. The character is framed with start, stop, and parity bits.

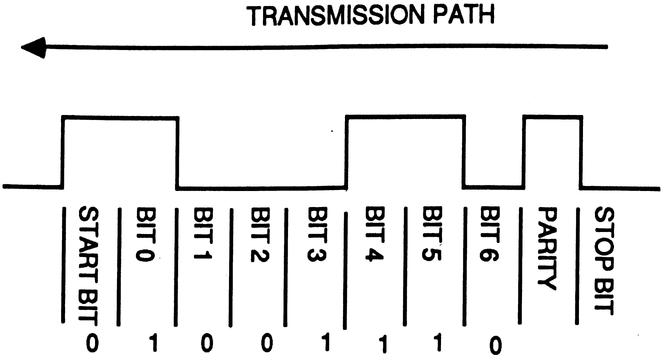


Figure 10-3 An example of serial data transmission.

Bit 0 is the first bit of the character to travel down the line to the receiving terminal. It is followed by bit 1, bit 2, bit 3, bit 4, bit 5, and bit 6. Bit 7 would follow if the data word were 8 bits long, instead of 7 bits. An optional parity bit can be specified as a part of the protocol and follows the data when parity is enabled (see parity discussion later in this chapter).

As data are transmitted one bit at a time, serial communication is slower than parallel transmission. The main advantage is the reduced cost of the transmission system and wire required to carry the signals. Additionally, serial communications lends itself nicely to transmission over common carrier networks such as the phone system or Telex network, whereas parallel communication does not. Your telephone does not have eight wires associated with transmitting voice over the network, it has only two or four wires. Therefore, it preludes the use of parallel transmission of data.

The UART

A device known as a UART (Universal Asynchronous Receiver Transmitter) converts the data received in parallel from the computer's bus to serial format required for telecommunications. The UART is a fairly sophisticated device which allows a great amount of flexibility in the character formats used in communications.

The UART is actually a dedicated microprocessor which has been designed to handle the communications interface and protocol. The UART is an integrated circuit consisting of a number of registers that are programmed to carry out the desired communication functions. The UART used on the Asynchronous Communications Adapter (ACA) in the IBM PC is the National Semiconductor INS 8250.

Table 10-1 lists the register ports of the ACA in the IBM PC. Programming the chip is fairly straightforward and is done for you initially by BIOS. However, you will have to establish the communications parameters as required by the specifics of your communications protocol. BIOS provides you with the necessary functions to accomplish this. I'll discuss protocol specifications in more detail when the programming example is presented.

Telecommunications

The serial bit stream from your computer, which is in digital form, must be converted to audio tones for transmission over the telephone system (Figure 10-4). Originally designed to handle only voice, which is in analog form, the conversion

Table 10-1
Serial Board Port Assignments
Port address assignments for the optional serial I/O (RS232C) boards found in the IBM PC.

Function	I/O	Board Number 1	Board Number 2
Transmitter Holding Register	OUT (Bit 7 of the line control register = 0)	03F8H	02F8H
Receiver Holding Register	IN	03F8H	02F8H
Baud Rate Divisor (LSB)	OUT (Bit 7 of the line control register = 1)	03F8H	02F8H
Baud Rate Divisor (MSB)	OUT (Bit 7 of the line control register = 1)	03F9H	02F9H
Interrupt Enable Register	OUT (Bit 7 of the line control register = 0)	03F9H	02F9H
Interrupt Identification Register	IN	03FAH	02FAH
Line Control Register	OUT	03FBH	02FBH
Modem Control Register	OUT	03FCH	02FCH
Line Status Register	IN	03FDH	02FDH
Modem Status Register	IN	03FEH	02FEH

process is still a prerequisite to communicating data over most telephone circuits. The conversion is performed by a device called a modem. These frequencies (or tones), relate to the data's binary equivalent, binary 1 or 0. The U.S. frequency assignments are shown in Table 10-2. By dividing the audio spectrum into separate frequency assignments, two-way communications over a single wire may take place.

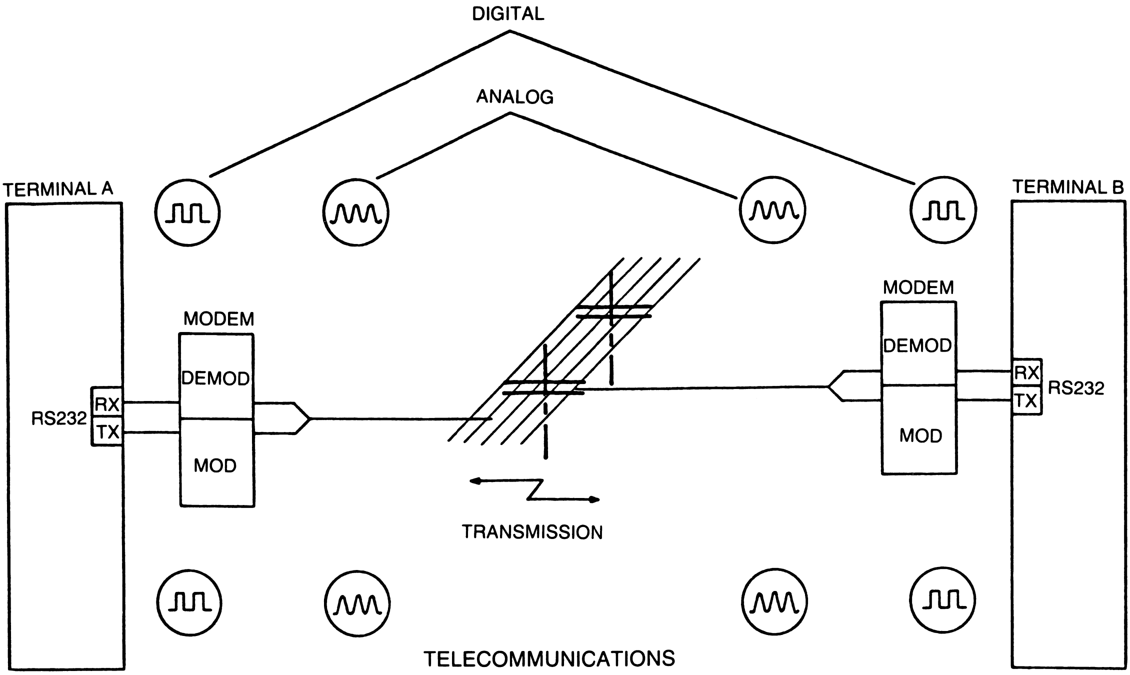

DIGITAL
ANALOG
TERMINAL A
TERMINAL B
MODEM
DEMOD
MOD
RS232
RX
TX
MODEM
DEMOD
MOD
RX
TX
RS232
TRANSMISSION
TELECOMMUNICATIONS

Figure 10-4A Digital to analog conversion of data via modems for transmission of data over common carrier facilities such as the telephone network.

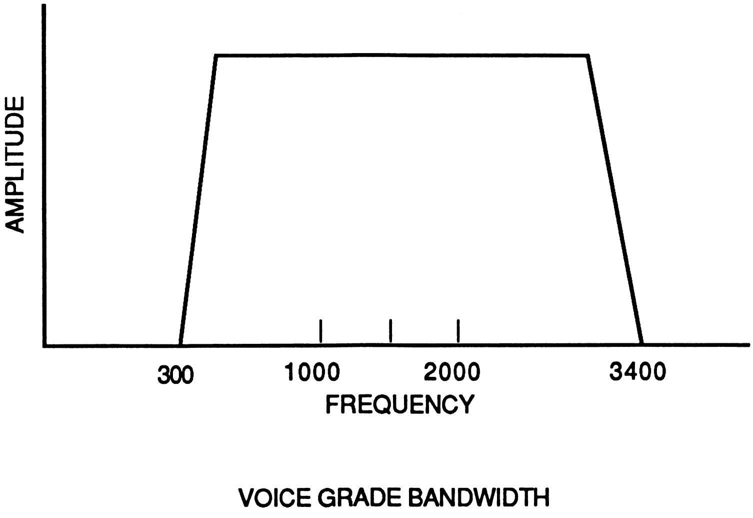
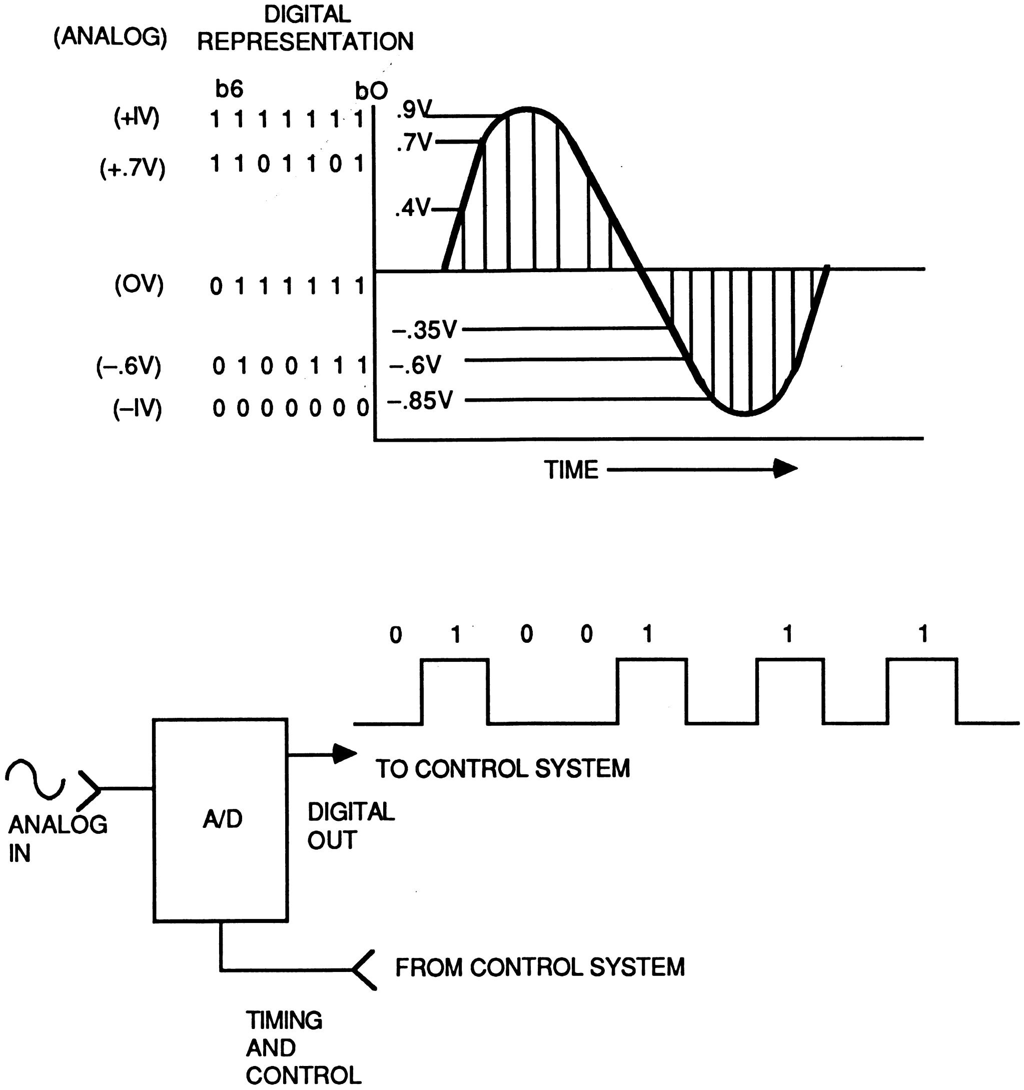


Figure 10-4B Range of audio frequencies allowed over the telephone network (voice bandwidth).


DIGITAL
(ANALOG)
REPRESENTATION
b6
b0
(+IV)
1 1 1 1 1 1 1
(+.7V)
1 1 0 1 1 0 1
(OV)
0 1 1 1 1 1 1
(–.6V)
0 1 0 0 1 1 1
(–IV)
0 0 0 0 0 0 0
.9V
.7V
.4V
–.35V
–.6V
–.85V
TIME
0 1 0 0 1 1 1
TO CONTROL SYSTEM
ANALOG IN
A/D
DIGITAL OUT
FROM CONTROL SYSTEM
TIMING AND CONTROL

RESOLUTION ≈ .0156V/BIT

Figure 10-4C Typical method of converting an analog signal to digital form is through the use of an Analog to Digital Converter (A/D or ADC).

Table 10-2
U.S. Frequency Assignments

Standard	Baud	Modulation Type	Transmit Space	Transmit Mark	Receive Space	Receive Mark
Bell 103	300	FSK Full Duplex	1070	1270	2025	2225
Bell 113	600	FSK Full Duplex	1070	1270	2025	2225
Bell 202	1200	FSK Half Duplex	2200	1200	----	----
Secondary Channel	5			387 Originate	----	---- Answer
Bell 212	1200	PSK Full Duplex	1200	2400	2400	1200

Also shown in Table 10-2 are the modulation type and duplex mode. For the Bell 212 modems, the modulation rate is half of the baud rate shown (e.g., 600), due to dibit encoding of the data. All transmit and receive frequencies are in hertz. (Table based on information found in Bell publications).

Methods of Analog Data Transmission

Analog signals are said to have a continuous range of possibilities, whereas a digital signal has two discrete possibilities. Figure 10-5 shows a typical analog signal, a sine wave. When used to communicate data, the sine wave is modulated by the data. Its amplitude, frequency, and phase are all capable of being modulated. Figure 10-6 shows the three fundamental types of modulation.

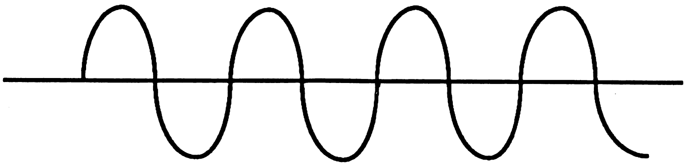


Figure 10-5

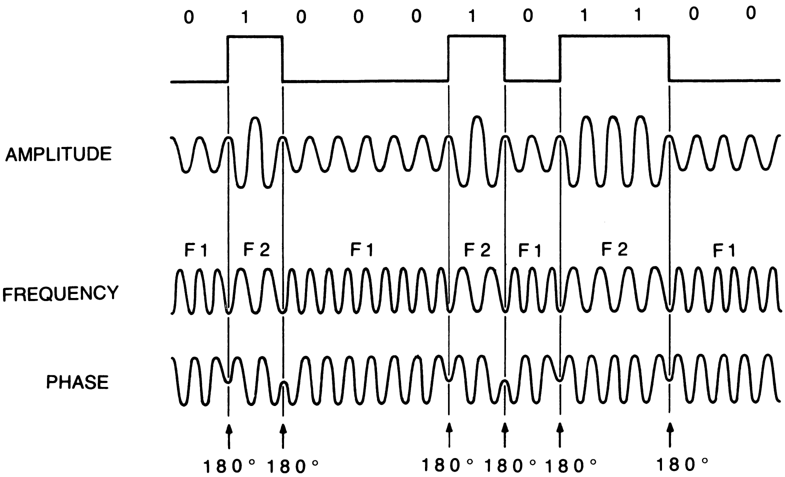


Figure 10-6 The three most common modulation techniques in use today. From top to bottom, amplitude modulation (AM), frequency modulation (FM), and phase modulation (PM). (Illustration based on the works of James Martin).

Frequency Modulation (FM)

Frequency Shift Keying (FSK)

FSK is a mode of FM whereby data is transmitted over telephone circuits by shifting the carrier frequency in response to a binary 1 or 0. The logic level (1 or 0) is determined by the data being sent. By allowing one frequency to designate a mark (binary 1) and another to designate a space (binary 0), digital data can be transferred from one point to another. The data is modulated at the sending terminal and demodulated by the receiving station. FSK is also referred to as F1, F2 transmission.

Phase Modulation (PM)

Phase modulation is used in higher speed transmissions (typically 2400 to 4800 bps). The phase of the waveform changes in response to binary 1 or 0. Table 10-3 depicts what is known as dibit phase encoding. Two bits are grouped together, dictating the number of degrees the carrier will change in phase. Receiving equipment interprets the phase shift and reinstates the proper binary grouping. Phase encoding is also used to group more than two bits together in transmission.

A grouping of 2 binary digits causes the carrier wave to shift in phase by the specified number of degrees. By grouping multiple bits of information to each

Table 10-3
DIBIT Encoding and Phase Modulation

2 bit binary grouping (DIBIT)	Phase Shift (degrees)
0 - 0	90
0 - 1	0
1 - 0	180
1 - 1	270

signal change, the modulation rate, or baud rate, can be lower than the total number of bits transferred in one second.

Amplitude Modulation (AM)

This technique varies the amplitude (voltage or current) of the carrier wave in response to the binary data. We are all accustomed to hearing this term used in conjunction with AM radio stations. The same technique is applied in data communications. Amplitude modulation requires more power and is more susceptible to noise than the two methods previously discussed.

Another aspect of the AM signal is the bandwidth, or range of frequencies the signal must occupy. For voice, the highest frequency used in telephony is approximately 3400 hertz (see Figure 10-4). A characteristic of AM signals is the generation of sidebands, or new frequencies formed by the mixing of the carrier wave and the modulating tone. Sidebands are created that are the sum and the differences of the modulating waveform. Therefore, the total bandwidth of an AM signal modulated by a voice is 8000 hertz, or twice the highest frequency used to modulate the carrier. AM systems are usually not employed in data communications.

A Closer Look at Duplex

Full Duplex

On a full-duplex communications channel, data transfers are allowed in both directions simultaneously. A four-wire circuit is most often used for full-duplex operation. Two wires carry data in one direction, while two others carry data in the opposite direction. Two-wire systems can be used if the audio spectrum is partitioned to provide separate receive and transmit frequencies. Figure 10-7 demonstrates how transmission over a 2-wire circuit takes place.

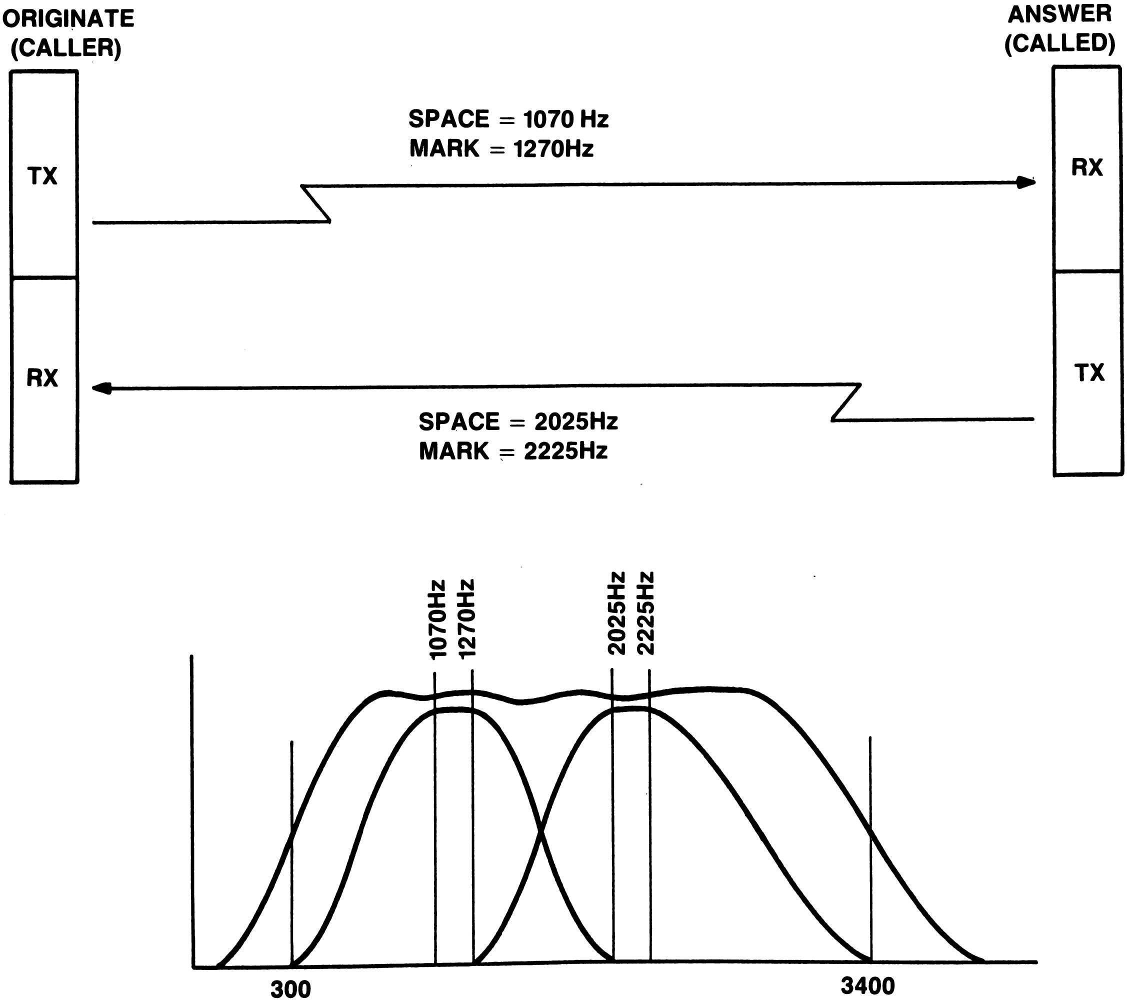

ORIGINATE
(CALLER)
ANSWER
(CALLED)
TX
RX
SPACE = 1070 Hz
MARK = 1270Hz
RX
TX
SPACE = 2025Hz
MARK = 2225Hz
1070Hz
1270Hz
2025Hz
2225Hz
300
3400

Figure 10-7 Full-duplex frequency assignments. By segmenting the available bandwidth of a given communications channel, a single wire can be used for full-duplex communications.

The transmit frequencies for the originating terminal are the receive frequencies for the answering terminal. The receive frequencies for the originating terminal are the transmit frequencies of the answering terminal. Bandpass filters restrict the bandwidth of each station's receiver, preventing one station from listening to its own frequencies. One station (or modem) must assume the originate mode, while the other terminal assumes the answer mode. If both stations were set to answer or both were set to originate, no information would be transferred.

Half Duplex

Half duplex is a communications mode whereby only one party can transmit at a time. When the transmission is complete, the other party can transmit, while the first can only receive. In order to accomplish true half-duplex communications, the transmission line must be turned around when it is the other party's turn to transmit. Line turnaround can take 250 milliseconds or more; therefore, the line is not utilized to its fullest capacity. Remember full duplex allows simultaneous 2-way transmission, without the need to turn the line around. There is no dead time while the transmission line must be turned around. Full-duplex operation, therefore, uses the transmission medium in a more efficient manner. The frequencies used in half-duplex and simplex operation are shown in Figure 10-8.

When operating a half-duplex channel, each terminal may locally echo each character it transmits. Local echoing to a CRT or printer is common in half-duplex systems. Since only one station is allowed to transmit at a time, there is no echoing of characters from the receiving terminal. It cannot transmit until the sending station is finished. Without a local echo option when communicating over a half-duplex channel, you would never see the characters you were transmitting.

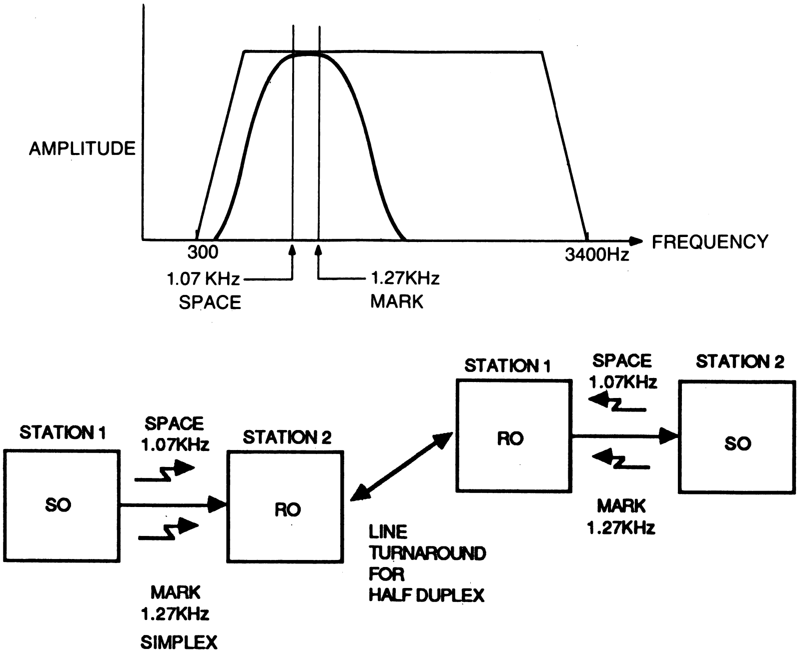

AMPLITUDE
FREQUENCY
300
3400Hz
1.07 KHz
SPACE
1.27KHz
MARK
STATION 1
SO
SPACE
1.07KHz
STATION 2
RO
MARK
1.27KHz
SIMPLEX
LINE
TURNAROUND
FOR
HALF DUPLEX
STATION 1
RO
SPACE
1.07KHz
STATION 2
SO
MARK
1.27KHz

Figure 10-8

In full-duplex operation, the host or main computer often echoes the characters back to your terminal as it receives them. If your terminal were to provide local echo, two characters for every one transmitted would appear on your CRT, the one the remote terminal sent back to you and the one your local echo function provides.

Simplex

Simplex operation (Figure 10-9) is a one-way-only data link. Two examples of simplex operation are line-of-sight communications and television signals that are broadcast to a satellite and retransmitted to a ground or earth station (receiver). Each path of the data link is one-way-only and constitutes a simplex channel. Keyboards, printers, and other receive-only or send-only computer peripherals are examples of simplex devices you may be more familiar with.

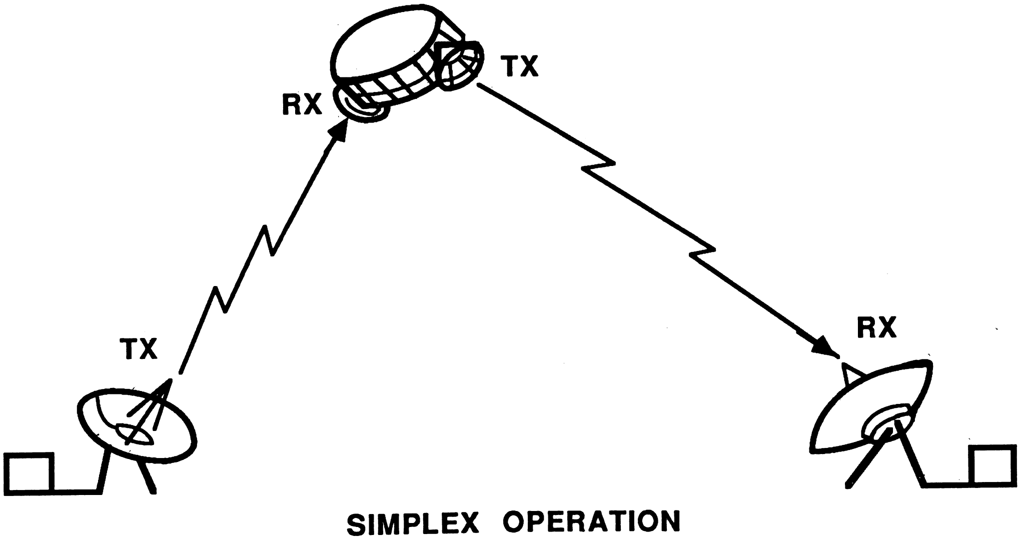


Figure 10-9

Asynchronous versus Synchronous Communications

Asynchronous Communications

Asynchronous transmissions are data transmissions which do not require a separate clock or other timing element to be sent by one terminal to synchronize reception by the other. Asynchronous communications are typically irregular (characters are transmitted on an irregular basis such as from a keyboard) and framed by a start and stop bit. Each character, as it is transmitted, has a start bit inserted in the data stream before the first bit (b0) is sent. After the last bit of the data word is sent, a stop bit of a specified interval is inserted.

During idle line conditions (when no characters are being transmitted), the line assumes the marking state, which is equivalent to a binary 1. The start bit is the reference element that notifies the receiving terminal a character is being transmitted. Being opposite that of the idle line condition, or binary 0, the start bit assumes what is known as the spacing condition. The receiving terminal interprets the data bits following the start bit as character data and looks for a valid stop bit designating the end of transmission. The stop bit may be of variable length.

For example, in 5-bit Baudot, the stop bit is 1.5 bits in length. It occupies one complete bit timing cell and half of another bit cell to signal the end of the character. This is the minimum amount of time the transmission line must be held in a marking condition between characters. If another character is not ready to be sent, the line is returned to idle and remains in the marking state until the next start bit occurs.

A disadvantage to asynchronous communications is its inefficiency when compared to synchronous forms of transmission. Since two or more extra bits are required in the transmission of each character, a great deal of time is spent sending framing information and not the actual data. For example, if transmitting standard 7-bit ASCII, with parity checking enabled and, with 1 start and 1 stop bit, 3 out of every 10 bits would be used to convey timing and error-checking information. That is, 30 percent of every character transmission time is used for supervisory functions.

Synchronous Communications

Synchronous communications can be thought of as a block oriented data transfer technique. An entire block of data is transferred at a time, instead of a single character as in asynchronous communications. Because transmissions occur without framing every character with start and stop bits, communications are faster and the transmission is more efficient.

Sending and receiving parties must be synchronized whenever communications are to occur. Since the data words or characters are not framed by synchronizing bits, the entire block being transferred is framed by synchronizing characters or flag characters. A clock or timing character must be transmitted when the line is idle to keep the communicating terminals in synchronization.

The ASCII control code SYN is often used to begin a frame or block of data. When the line is idle, SYN (SYNchronize) is transmitted. This control code allows the receiving terminal to synchronize to the data frame that follows. It starts a clock in the modem or receiver of the receiving terminal, which locks in on the control and data characters, or bits following the SYN character. An example of a synchronous frame is shown in Figure 10-10.

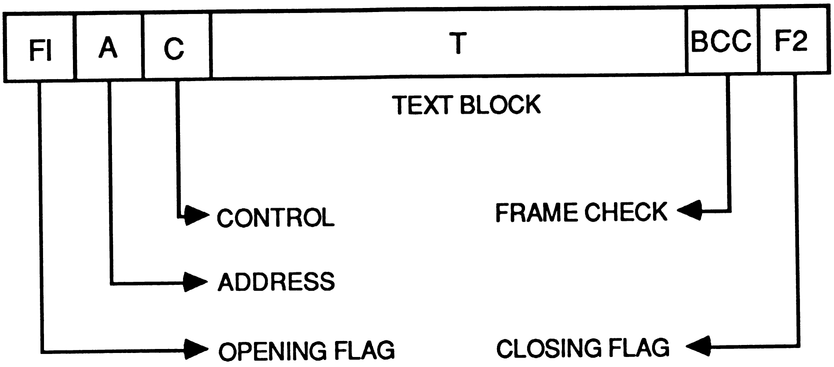


Figure 10-10 A frame of information as formatted by synchronous data line control communications.

Each frame begins with an 8-bit flag sequence of binary 01111110. The frame ends with a flag of the same binary representation. The address and control fields are each 8 bits in length. The text block must be a multiple of 8 bits and is followed by a frame check sequence that is 16 bits in length. The frame ends with an 8-bit flag sequence.

The frame is divided into fields, each carrying specific information about the text or the communications in progress. For example, a header may precede the text that not only contains the SYN character but also may contain address information, the length of the block being transmitted, and a block number identifying which block in a series of blocks is being transmitted.

The trailer usually contains error-checking information, such as a checksum which is generted on the body of the block or a Cyclic Redundancy Check (CRC) of the block. It may also contain a closing byte or flag. SDLC opens a block with the binary flag 01111110 and closes the block with the same flag.

The receiving terminal keeps track of the current block number and verifies the error-checking codes received in the frame. If it detects an error, such as the wrong checksum, it sends a Negative AcKnowledgement (NAK) to the sending terminal. The NAK may be in the form of a block, complete with header and BCC, or the ASCII control code NAK may be all that is sent to the other computer.

The NAK tells the other terminal to retransmit the last block (or blocks) of information, because they were received in error. However, if the receiving computer signals back with the control code ACK (ACKnowledgment), it is saying that it found no errors, and it agrees with the checksum or CRC that was received. This handshaking continues until all blocks are transferred. Usually the code EOT (End Of Text) signals the end of the transfer.

There are several high level protocols associated with synchronous block oriented data transfers. They also provide more sophisticated error checking than parity checking allows for in asynchronous communications. Some of the more popular synchronous protocols in use today are HDLC (CCITT and ISO recommendations), ADCCP (Advanced Data Communications Message Protocol, defined by ANSI), BISYNC (Binary Synchronous Communications Protocol), SDLC (Synchronous Data Link Control, developed by IBM), and DDCMP (Digital Data Communications Message Protocol, developed by Digital Equipment Corp.—DEC).

SDLC is a bit oriented protocol. The individual bit positions within the frame take on significance as to the information they represent. Bits 0–7 are always the opening flag character, and bits 8–15 always contain address information as to the frame's origin or destination. The significance these bit positions represent never changes when using this protocol. Other protocols may be classified as byte or character oriented in nature.

Since a block of data can consist of multiple bytes, or characters, and framing occurs only once for each block, line efficiency is increased over asynchronous protocols. As an example, let's pick a block length of 1024 characters each consisting of 7 bits (7168 bits). If we assume 16 bits for beginning and ending flags, 16 bits for the address and control fields, and a 16-bit frame check sequence, only 48 bits of the total 7,216 bits are used for control information. Only .66 percent is spent sending framing and nondata oriented information.

Baud Rate

Baud rate is probably the most misunderstood term in telecommunications. What baud rate describes is the modulation rate of the data being transferred. It is usually equated with data being transferred in Bits Per Second (BPS). Baud rate actually is equal to the total number of signal elements per second. If 300 signal elements are sent in one second, you are transmitting at 300 baud.

There are other techniques in data communications that allow more than 1 bit to be transmitted each time the signal changes. Therefore, the baud rate could be less than the bits per second. By grouping binary digits into dibits and using phase shifts to modulate the carrier, each signal change would correspond to 2 bits. The baud rate would be half the bps rate of the transfer. Table 10-3 illustrates the use of dibit encoding to phase modulate the carrier.

Baud rate can only be equated to bits per second when a single binary digit (bit) corresponds to a change in signal. In most low speed communications (300 bps to 1200 bps), the modulation technique used is FSK (Frequency Shift Keying), where bps is equal to baud.

Error Checking

With any type of communications, error checking is extremely important. If the ability to insure the data integrity of the channel is left to chance, it could easily ruin or seriously degrade our communications capability. Noise is an ever-present companion to any electrical signal. Not only are transmitted signals susceptible to noise, they also are a source of noise to other transmissions.

This type of noice source is commonly referred to as crosstalk. It's evident when you place a telephone call to your long-lost brother, and you can clearly understand another conversation being carried on another circuit. A signal generated from another source is interfering with signals generated by you and your brother. Noise on telephone circuits can be caused by telephone switching gear, atmospheric conditions, or circuits in close proximity to each other. The faster the data transfer rate, the more bits or characters will be lost due to noise.

Another type of noice source is the impulse or random source. It can be likened to a noisy car randomly driving by your house (see Figure 10-11). If, in the middle of a conversation you're having with a neighbor, the car backfires, a portion of the conversation may be lost. If the car suffers from some other abnormality, such as a missing muffler, many words may be lost. In data communications, this type of impulse noise is referred to as burst noise. The higher the communications rate, the more serious the problem. Rather than losing one or two bits (words, in our human to human analogy), entire characters may be lost due to bursts of noise on a data communications channel.

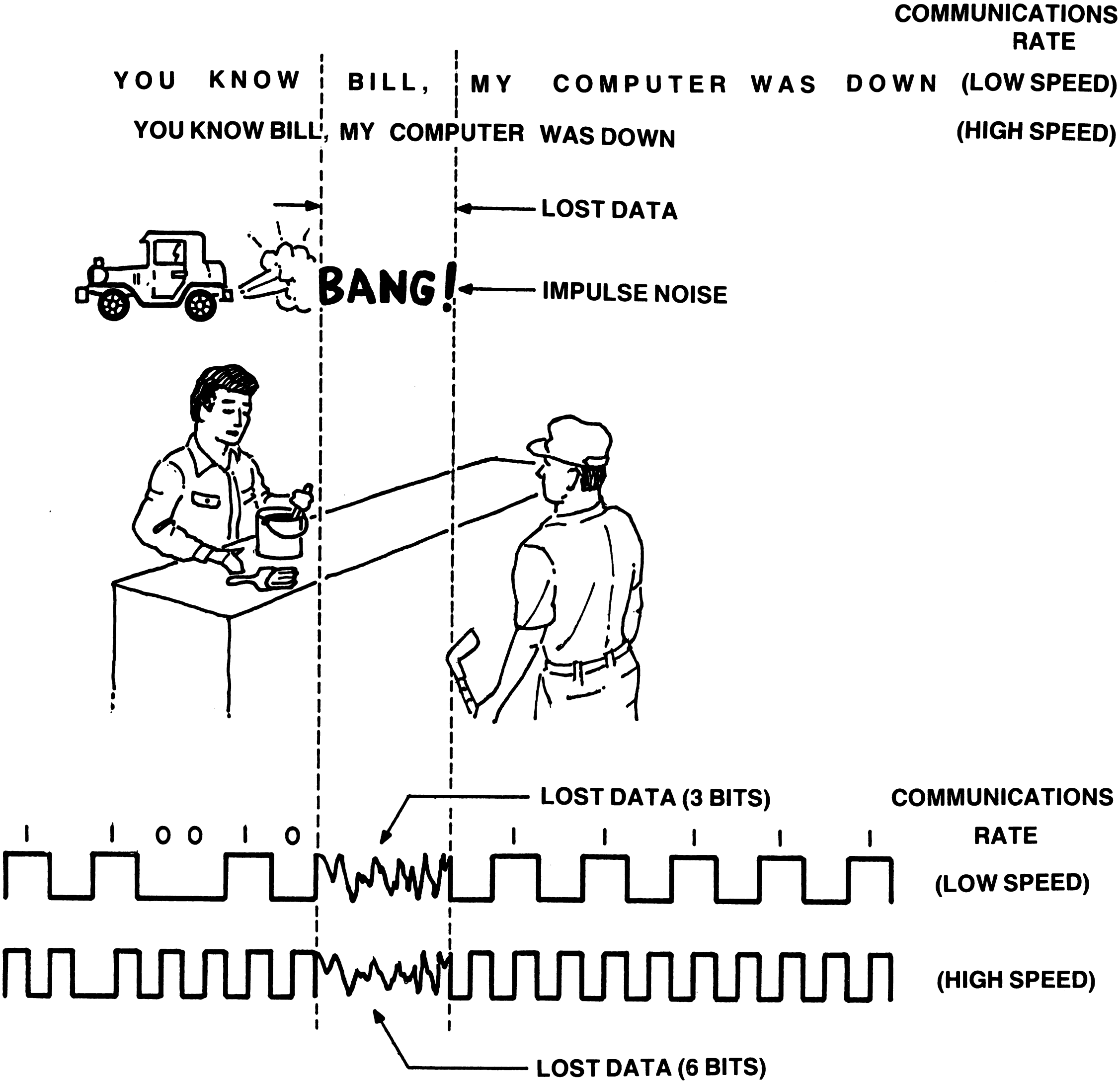

COMMUNICATIONS RATE
YOU KNOW BILL, MY COMPUTER WAS DOWN
(LOW SPEED)
YOU KNOW BILL, MY COMPUTER WAS DOWN
(HIGH SPEED)
LOST DATA
BANG!
IMPULSE NOISE
LOST DATA (3 BITS)
COMMUNICATIONS RATE
1 1 0 0 1 0 1 1 1 1 1
(LOW SPEED)
(HIGH SPEED)
LOST DATA (6 BITS)

Figure 10-11

There are other contributing factors to the degradation of transmitted signals aside from the noise that affects communications. Electrical properties of the transmission medium can distort the signal to a point where it is rendered useless. Signal repeaters capable of regenerating the signal must be located at reasonable distances from the transmitting source. When the signal is regenerated, the signal's level (amplitude), phase, and frequency characteristics are restored, noise is removed, and the signal is retransmitted to the next repeater or to the final destination.

Parity

Parity is an error-checking technique associated with asynchronous communications. It is an optional bit inserted into the data stream that is dependent on the total number of bits set to a logic 1 in the data word. The total 1 bits, therefore, may be an odd or even number.

The examples shown in Figure 10-12 demonstrate how parity is determined. If odd parity is being used, the total number of logic 1 bits must be an odd value. Conversely, the total number of logic 1 bits must be even if even parity is used.

When parity is enabled in asynchronous communications, the total number of binary ones that occur within the word must be even or odd. The state of the parity bit is dependent upon the parity type you choose. It is set to a binary one whenever the total number of binary 1's in the word does not equal the parity mode (even or odd) that you have chosen. Examples are shown for both even and odd parity in Figure 10-12.

Assume for a moment you are communicating with odd parity. The binary representation of the ASCII letter A is 1 0 0 0 0 0 1. There is an even number of binary ones in the code; therefore, the parity bit would be set to a binary 1 to make the total an odd number. The ASCII representation for the letter C is 1 0 0 0 0 1 1 (43 hex) and contains an odd number of binary ones. Thus, the parity bit is reset to zero for this character.

Some terminals use what is referred to as marking or spacing parity. In this case the parity bit is always set to a mark or space. It will never vary.

ASCII Character	Binary	Parity Bit ODD	Parity Bit EVEN
A	1000001	1	0
B	1000010	1	0
C	1000011	0	1
D	1000100	1	0
E	1000101	0	1

Figure 10-12 Examples of the use of parity

Other error-checking techniques are more elaborate. A Cyclic Redundancy Check (CRC) is often appended to a synchronous frame. The CRC is generated by multiplying the transmitted data against a known polynomial. I refer you to the bibliography for additional reading material on higher level error-checking and correction techniques.

Modems

Modem stands for MOdulator/DEModulator. A modem modulates or demodulates an analog signal for data communications. The telephone system is designed to accept an analog signal (your voice) and transmit it to a central office for further conditioning before sending it on to the party at the other end of the line. Being analog in nature, the system requires the data be converted from your computer's digital signals into analog form before it is transmitted over common carrier lines. The modem does this.

Most low speed modems (up to 1200 bps) use Frequency Shift Keying (FSK) to translate digital binary to audio frequencies within the bandwidth of the telephone. The telephone company limits the bandwidth allowed for voice transmissions over the switched public network to between 300 and 3400 hertz. By using different frequencies within this spectrum, each modem is able to generate or recognize a binary 1 or 0. Table 10-2 shows the accepted US standard frequencies for binary transmission.

By dividing this band into smaller bands of frequencies, two-way transmission can occur over a two-wire system (one wire for signal transfer and one wire for common ground). The operating mode of each modem (originate or answer) dictates which set of frequencies is used for transmit and receive.

The frequencies are termed mark (binary 1) and space (binary 0). The sending modem modulates the carrier by shifting its frequency, in response to the binary equivalent presented to its transmit data input. The receiving modem demodulates the signal by converting the frequency shift to the binary equivalent required by your terminal.

Most modems are interfaced to a computer or terminal by connection to an RS-232C interface. This interface is a well-defined standard for interconnecting Data Terminal Equipment (DTE) and Data Circuit terminating Equipment (DCE). Recently, there has been a number of computers boasting of internal, or Direct Distance Dial (DDD), modems. All that is required is to connect the telephone cable to the RJ-11 jack of the computer. The modem is built into the unit, as is the telephone interface. Single chip modems are now available from a number of manufacturers. Texas Instruments, Motorola, and Advanced Micro-Devices produce single chip modems. The chips still require support circuitry, such as FCC approved data access arrangements, which connect the modem to the phone.

An acoustic coupler is an interface consisting of a modem, but it is coupled to the telephone system acoustically. It contains a microphone and speaker that interface to the telephone without any wires being connected to the system. Most acoustic couplers operate at 300 baud or less, but there are some available that allow operation at baud rates of up to 1200 baud. To establish a call using an acoustic coupler, first the call is manually dialed. When the answering tone is heard through the telephone receiver, the handset is placed into the cradle of the coupler, and the coupler is placed in originate mode. The microphone and speaker of the coupler transmit and receive the frequencies required for transmission to and from the handset acoustically.

The RS-232C Standard

The RS-232C Standard (International Standards CCITT V.24 and V.28), developed by the Electronics Industry Association (EIA), defines pin assignments for the connection of Data Terminal Equipment (DTE) with Data Circuit terminating Equipment (DCE). The standard defines the electrical, mechanical, and functionality of interconnecting circuits. Copies of the standard may be purchased from the EIA at 2001 Eye St., N.W., Washington, D.C. 20006.

Table 10-4 contains the signals and pin numbers where the signals appear on a standard RS-232C connector. Although the most popular connector is the DB-25 connector (as used on the IBM PC asynchronous communications adapter), the standard does not specify the actual connector to be used. The international standard IS2110-1980 does specify the 25-pin connector and pin assignments. In their forward to the RS-232C standard, the EIA points out there are some incompatibilities between the RS-232C and the ISO (International Standards Organization) standard.

Whatever connector is used, the signals should be used in a uniform manner and appear on the designated pins. You should also be aware that the received line signal detector signal is commonly referred to as Carrier Detect (CD). It signals to the DTE that a carrier has been detected by the DCE. If the signal is in the off state (mark), then the DCE is either not receiving a valid carrier, or it is receiving one unsuitable for demodulation.

Table 10-5 contains the electrical and logical conditions the circuits assume. Interchange voltage refers to the voltage present at the interface connector pin for a given circuit. For example, the circuit DTR (Data Terminal Ready) is on pin 20 of the connector. The standard defines the circuit as being active (on), when the voltage is positive (between +3 volts and +25 volts). DTR would be in what is

Table 10-4
RS232C Signal and Pin Assignments

Reprinted with permission from the EIA's RS-232C standard.

Pin Number	Circuit	Description
1	AA	Protective Ground
2	BA	Transmitted Data
3	BB	Received Data
4	CA	Request To Send
5	CB	Clear To Send
6	CC	Data Set Ready
7	AB	Signal Ground (Common Return)
8	CF	Received Line Signal Detector
9	—	(Reserved for Data Set Testing)
10	—	(Reserved for Data Set Testing)
11	—	(Unassigned)
12	SCF	Sec. Rec'd. Line Sig. Detector
13	SCB	Sec. Clear To Send
14	SBA	Sec. Transmitted Data
15	DB	Transmission Signal Element Timing (DCE Source)
16	SBB	Secondary Received Data
17	DD	Receiver Signal Element Timing (DCE Source)
18	—	Unassigned
19	SCA	Secondary Request To Send
20	CD	Data Terminal Ready
21	CG	Signal Quality Detector
22	CE	Ring Indicator
23	CH/CI	Data Signal Rate Selector (DTE/DCE Source)
24	DA	Transmit Signal Element Timing (DTE Source)
25	—	Unassigned

termed as the spacing condition, with a binary equivalent of 0. When put in the marking state, the circuit's voltage is negative (−3 volts to −25 volts), the binary equivalent of 1. All signals follow this convention. Furthermore, the drivers associated with each signal must be able to withstand short-circuit conditions on their outputs.

Using the IBM PC as an example (see the *IBM Technical Reference Manual*), DTR is presented to the DB-25 connector through U12, an RS-232C driver. The driver performs a voltage shift and inversion on the TTL level data presented to its input. Therefore, to properly set DTR to the active condition on this computer, it is necessary to set the control bit for DTR to TTL logic 0 (0 volts). Due to the voltage translation and inversion (0 volts to +12 volts) of U12, the signal is in the spacing or

Table 10-5
RS232C Signal Functionality

Notation	Interchange Voltage	
	Negative	Positive
Binary State	1	0
Signal Condition	MARKING	SPACING
Function	OFF	ON

on condition. If a logic 1 is output to the driver, the circuit assumes the off condition. Table 10-6 depicts signals used on the IBM PC for data communications.

Terminals can communicate up to 50 feet using the RS-232C interface and at speeds up to 19,200 bps. I'll discuss the RS-232C and its implications in the following text. You are advised to write to the EIA at the address given above and obtain copies of the standard. Related EIA interface standards are the RS-422, RS-423, and RS-449. The RS-449 standard permits transmission speeds up to 2 million bits per second and cable lengths of up to 60 meters.

Table 10-6
IBM PC RS-232C pin assignments

Pin Number	Signal	
1	PGND	(Protective Ground)
2	TD	(Transmit Data)
3	RD	(Receive Data)
4	RTS	(Request To Send)
5	CTS	(Clear To Send)
6	DSR	(Data Set Ready)
7	SGND	(Signal Ground)
8	CD	(Carrier Detect)
20	DTR	(Data Terminal Ready)
22	RI	(Ring Indicator)

Figure 10-2 depicts the RS-232C signals used in the IBM PC.

Computer to Computer Communications

Most of the previous discussions have revolved around communications over the telephone. Computer to computer communications are also possible with the same software used for telecommunications. A special cable is required to interconnect two computers using the RS-232C interface. The term null cable is used to refer to such a cable. In a null cable, pins 2 and 3 (TD and RD) are exchanged (see Figure 10-3). Other signals may need to be tied together, such as DTR with DSR and RTS with CTS at each connector. Depending on what the software requires (i.e., CTS active, etc.) to allow communications, these signals or others are usually tied to their counterparts at the connector of each terminal to give the appearance of functionality to the software.

For example, connecting CTS to RTS at one terminal would give the appearance of CTS being true. RTS is the signal the terminal would be sending to a modem if one were connected. CTS would be the signal returning to the terminal from the modem. By tying these two signals together, the terminal, on activating RTS, would see CTS being returned, or also in the on condition. Nearly every application is different, requiring different signals of the RS-232 to be present to fool the terminal's software.

Not all communications programs fully utilize all of the signals provided by the RS-232C. Referring to Table 10-6, you'll see that not all of the RS-232C signals are brought into the IBM PC. All of the signals are not required for most communications. Even the signals available are not all used in all communications software.

Figure 10-13 shows a wiring diagram for a null modem cable that can be used to transfer data between two computers. By using the cable and the appropriate software, the two computers are able to share information and resources, such as disks and printers. You are restricted to a communications distance of less than 50 feet when communicating through the RS-232C. This is not usually a problem, but you should be aware of the limitations of the standard. Other communications standards define communicating distances up to 4000 feet. (RS-422).

The Program: COMM.ASM

As you can see from the source listing of COMM.ASM in Listing 10-1 (Appendix D), the program is broken down into small modules. This allows you to change or adapt the modules relatively quickly if more features are desired or the system's hardware or operating system should change. COMM.ASM uses several system calls within its framework. Should changes occur to the operating system, COMM.ASM can easily adapt to its new environment.

Because the program consists of many modules, it is necessary to provide you with some introductory material on how the modules interact. Some are used only to change an operational parameter, as in the terminal characteristics of the program. Still others are integral parts of the program that are executed on every pass of the job list.

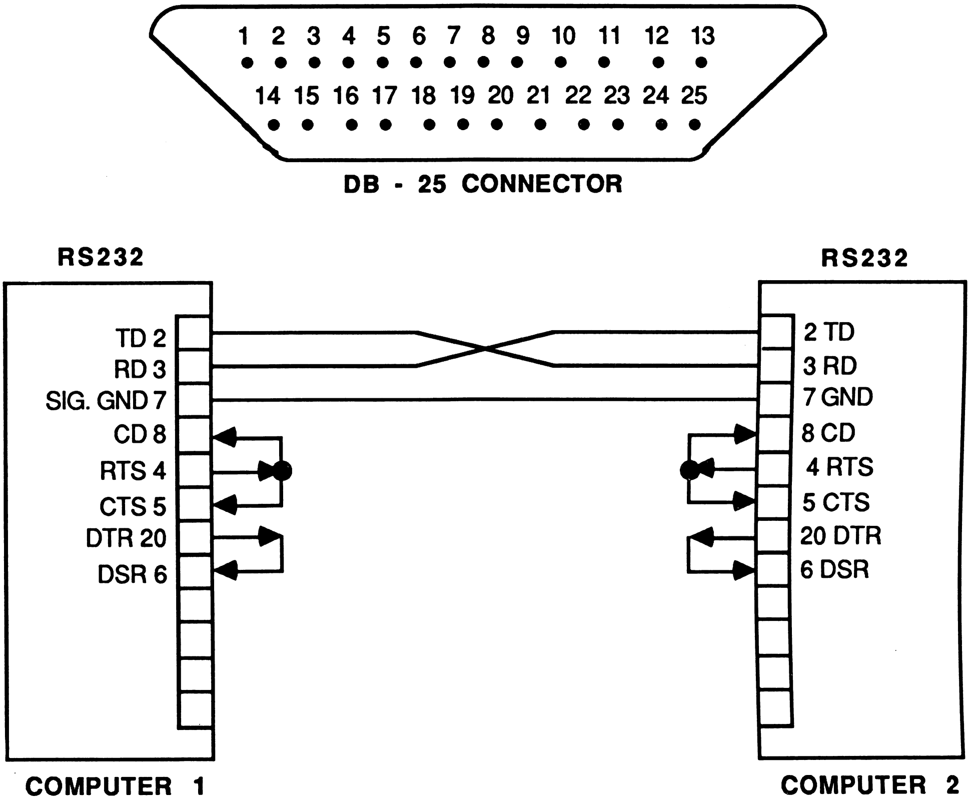

1 2 3 4 5 6 7 8 9 10 11 12 13
14 15 16 17 18 19 20 21 22 23 24 25
DB - 25 CONNECTOR
RS232
RS232
TD 2
RD 3
SIG. GND 7
CD 8
RTS 4
CTS 5
DTR 20
DSR 6
2 TD
3 RD
7 GND
8 CD
4 RTS
5 CTS
20 DTR
6 DSR
COMPUTER 1
COMPUTER 2

Figure 10-13 Null modem cable wiring diagram

Why Not MS-DOS Serial I/O?

Two methods that can be used to receive and transmit characters via the asynchronous communications adapter are BIOS and MS-DOS function calls. The third method deals directly with the communications port and reduces the portability of the program to other MS-DOS computers. While it may not always be desirable to bypass the operating system and deal directly with the port, useful and sophisticated communication programs must be written in this manner.

As I stated in the last chapter, the functions provided under MS-DOS for serial I/O are minimal at best and not well suited for generalized communications. Since MS-DOS links to the BIOS routines, the BIOS routines also prove to be inadequate when using those skeletal functions that MS-DOS does provide.

If you inspect the BIOS listing in the *IBM Technical Reference Manual* on pages A-22 - A-25, you'll find that DSR and CTS must be asserted before transmitting a character over the communications channel. While the protocol of waiting for these signals to be asserted is technically correct, most systems give you an option as whether or not the protocol is desired. This type of protocol is often referred to as electrical pacing. It is used to pace the communications line, so that if the DCE is not prepared to receive characters from the DTE, the transmissions from the DTE will be temporarily suspended. Many systems and lower-priced modems do not use these signal lines. Therefore, you must prepare a null modem cable, as previously described, to make the software believe that these signals are active.

The BIOS routines do not afford the necessary control over the communications channel when operated in an interrupt driven mode. The use of interrupts is desirable when communicating with another computer. If interrupts were not used, the communications port would have to be periodically polled to insure the proper reception of received characters. If the communications channel is not polled frequently enough, characters may be lost. When the UART receiver holding register is full (a character has been received) and another character arrives before the program retrieves the first character, the UART generates an overrun error. Therefore, when using higher communication rates (1200 to 9600 baud), your program must be ready to accept a character as soon as it is available at the receiver port.

This severely limits the amount of processing that can take place, since most of the 8088's time is spent polling the communications channel. The use of interrupts allows the 8088 to perform whatever processing chores are required without spending valuable time polling the channel. It also allows higher baud rates to be used in communications without sacrificing processing time dedicated to other tasks.

In order to carry out communications via the asynchronous communications adapter in the most efficient manner, you must directly program and access the registers of the National Semiconductor INS 8250 UART and the Intel 8259 programmable interrupt controller.

BIOS routines can be used to initialize many of the communications parameters. However, the registers of the chips must be accessed to perform other functions. I'll first discuss the registers of interest to us and then explain what MS-DOS and BIOS routines are acceptable when programming for data communications.

Where to Start

Refer to Listing 10-1 as I discuss COMM.ASM's operation. The listing contains bit map assignments to the 8250 UART registers I'll describe in a moment.

Physical Port Assignments

If you look at Listing 10-1, the port assignments for the National Semiconductor 8250 UART found on the asynchronous communications adapter board are listed at the beginning of the source file.

The Baud Rate Divisor Registers

Type: **Output**

Ports: **03F8H (LSB of divisor word)**
03F9H (MSB of divisor word)

To access the baud rate divisor registers, bit 7 of the line control register must be set to a binary 1. The least significant byte can be accessed at port 03F8H, and the most significant byte at port address 03F9H. These registers specify a divisor that determines the baud rate for the communications channel. The 1.8432-megahertz clock signal is divided by the value specified in the divisor registers to generate the necessary baud rate timing.

Notice that simply dividing the 1.8432-megahertz clock signal by the divisor specified does not return the baud rate. The value returned by performing this division is 16 times the baud rate. The 8250 must sample the Receive Data input (RD) at a rate of 16 times the baud rate to be sure the data transitions (mark to space, etc.) are valid. On seeing a mark to space transition, the UART samples the RD line toward the middle of the bit cell, or on the 8th of the 16 clocks, to verify that the transition was caused by a valid start bit and not by noise being generated on the line. If the RD input is still in a spacing condition, the initial mark to space transition is then assumed to be generated from a valid start bit. Therefore, you can calculate the baud rate divisor by 1,843,200/(BAUD RATE x 16). For those who do not have a calculator handy, use the table shown in Listing 10-1.

You can program the channel for 9600-baud operation in the following manner:

```
MOV  AL,80H         ;Set bit 7 of line control register.

MOV  DX,03FBH       ;Port address for line control register.

OUT  DX,AL          ;Bit 7 now set.

DEC  DX             ;Point to port 03F9H, MSB of divisor.
```

continued

```
DEC   DX               ;
MOV   AL,00H           ;MSB of divisor
OUT   DX,AL            ;Set up MS of divisor.
DEC   DX               ;Point DX to port 03F8H
MOV   AL,0CH           ;Get LSB of divisor
                       ;(1.8432 Mhz / (9600 * 16)) = 12 = 0CH
OUT   DX,AL            ;Set LSB of divisor.
```

Once the baud rate divisor is set, the remainder of the communication parameters can be specified.

The Line Control Register

Type: **Output**

Port: **03FBH**

The line control register must be programmed to the communications parameters desired. You have control over the word length, number of stop bits, and whether to enable odd or even parity or disable parity altogether.

Bit 6 can be used to transmit a break on the communications channel. A break is a continuous space. A line break may signal the other system to stop transmitting, because there is an abnormality at the other end of the line.

When bit 7 in the line control register is reset, the transmitter and receiver holding registers at port 03F8H and the interrupt enable register at port 03F9H can be accessed. When bit 7 is set to a binary 1, those two port addresses provide access to the baud rate divisor registers. The least significant byte of the baud rate divisor is accessed via port 03F8H, and the most significant byte can be accessed via port 03F9H.

For example, you would output a value of 0FH to port 03FBH to program the chip for 8-bit word lengths with odd parity and 2 stop bits. Bit 7 is now reset and allows the receiver and transmitter holding register to be accessed rather than the baud rate divisor registers.

Transmitter and Receiver

Type: **Input (Receiver)**

Type: **Output (Transmitter)**

Port: **03F8H**

The transmitter holding register (THR) and Receiver Holding Register (RHR) are both assigned to port 03F8H. Data is read from the RHR by using the statements:

```
MOV   DX,03F8H           ;Get the port address of the RHR.

IN    AL,DX              ;Get data from port.
```

Similarly, data is transmitted by using the statements:

```
MOV   DX,03F8H           ;Get the port address of the THR

OUT   DX,AL              ;Transmit the character.
```

Each of these registers is capable of generating an interrupt. The transmitter can interrupt the 8088 when the holding register or shift register is empty, and the receiver can interrupt the 8088 when a character has been received (RHR full), or when a receive error has occurred.

The Modem Control Register

Type: **Output**

Port: **03FCH**

The modem control register controls the setting of the modem control signals DTR and RTS. If b0 = 1, then DTR is active (space), and, if b1 is set to a binary 1, RTS is set to the on state (space). Two other signals generated by the 8250 are controlled through this register, OUT1* and OUT2*. The OUT2* signal is active when b3 in the modem control register is set to a binary 1. Bit 3 must be set if interrupts are enabled from the ACA board. OUT2* enables the gating of the interrupt signal to the system bus of the PC. You can see this pictorially in the schematic on page D-88 of the *IBM Technical Reference Manual* (publication 6025005).

When bit 4 is set, the 8250 is put into a special diagnostic mode. The receiver input and the transmitter output are connected in the 8250. Any characters transmitted are therefore received. The physical receiver connection to the communications line from the chip is disconnected, and the transmitter output is set to a marking condition. The diagnostic mode is useful to test the receive and transmit functions of the 8250 to test interrupt service routines.

The remaining bits, b5, b6, and b7, in the modem control register must always be set to zero.

Interrupt Enable Register

Type: **Output**

Port: **03F9H**

The interrupt enable register dictates what conditions (if any) will generate an interrupt to the 8088. Bit 3 of the modem control register must be set to a binary 1 if any of the interrupt types are enabled. Bits 4–7 of this register must always be set to zero. If bit 3 is set, the 8250 generates an interrupt when there is a change in the modem status register. The modem status register will be discussed in a moment.

If bit 2 is set, then the 8250 generates an interrupt when a break is detected on the receiver input or when a character is received in error. When bit 1 is set, the 8250 generates an interrupt when the THR is empty and ready for the next character to be transmitted. When bit 0 is set, it enables interrupts to be generated whenever a character has been received (RHR full).

The interrupt service routine determines which of the possible interrupts occurred. This can be easily accomplished by reading the interrupt identification port 03FAH.

The Interrupt Identification Register

Type: **Input**

Port: **03FAH**

To check whether an interrupt has occurred, test b0 of the interrupt ID port. If the bit is set, then there is an interrupt pending. If the bit is reset, no interrupts are pending. Bits 1 and 2 code the interrupt type and priority of the interrupts as follows:

b2	b1	Type	Priority Level
0	0	Change in Modem Status	3 (lowest)
0	1	THR Empty	2
1	0	RHR Full	1
1	1	Received error, or break detected.	0 (highest)

You must read this port after each interrupt has been serviced and prior to returning from the current service routine to see if any other interrupts occurred during the service routine. Failure to do so will inhibit interrupts which occurred during a service routine from being recognized.

The interrupt ID bits are reset when the appropriate action has been taken. For example, if a THR empty interrupt has occurred, the interrupt is reset when a character is output to the THR. To reset the receiver full interrupt, input the character from the RHR. Similarly, reading the line status clears level 0 interrupts, and reading the modem status register clears level 3 interrupts.

The Line Status Register

The line status register provides information about the communications channel. The register can be read to determine if a character has been received or to determine what type of error occurred, if any.

The port maps the status of the communications channel as follows:

```
b0 = 1 = Receive data ready

b1 = 1 = Overrun error

b2 = 1 = Parity error

b3 = 1 = Framing error

b4 = 1 = Break interrupt

b5 = 1 = THR empty

b6 = 1 = Transmit shift register empty

b7 is always zero.
```

You must check bit 5 before attempting to transmit a character. If b5 = 0, then do not output the character to the THR. Similarly, b0 should be checked prior to attempting to input a character from the RHR.

Overrun Error

If you have ever seen a baseball game when one base runner overruns the lead runner (also known as bush league base running), you can easily visualize what happens when a character overrun occurs in the receiver. The UART receives a character, and, before the program has a chance to retrieve the character, another character arrives at the receiver. The first character has been overrun by the second. This type of error normally does not occur in a properly designed program that uses interrupts.

Parity Error

A parity error occurs when parity error checking has been enabled, and the parity of the received character does not match the parity type in use. If the UART detects odd parity when even parity is enabled (or even when odd is enabled), a parity error is generated.

Framing Error

This type of error is caused by an improper character framing. Each character transmitted asynchronously is framed by start and stop bits. The stop bit returns the communications channel to a marking condition. A framing error is most likely to occur when the line does not return to a marking state after the character or parity bit has been transmitted. This condition can signal the start of a received break.

Modem Status Register

Type: **Input**

Port: **03FEH**

The modem status register returns the status of the signals that communicate with the RS-232C interface. The signals are used as modem controls. The port is bit mapped as follows:

```
b0 = 1 = Change in clear to send (CTS)

b1 = 1 = Change in data set ready (DSR)

b2 = 1 = Change in ring indicator (RI) (trailing edge from 1 to 0)

b3 = 1 = Change in carrier detect (CD)

b4 = 1 = Clear to send (CTS)
```

```
b5 = 1 = Data set ready (DSR)

b6 = 1 = Ring indicator (RI)

b7 = 1 = Carrier detect (CD)
```

The 8259 Interrupt Controller

Command Register

Type: Output

Port: 020H

Interrupt Mask Register (IMR)

Type: Output

Port: 21H

In a communications program that uses interrupts, it is necessary that the 8259 interrupt controller be set up to recognize the interrupts generated by the ACA board. The 8259 interrupt controller can generate up to 8 vectored interrupts to the 8088. The 8259 accepts Interrupt ReQuests (IRQs) from the peripherals attached to the system, determines which has the highest priority, and interrupts the 8088 CPU. A vector address is placed on the bus that the 8088 uses to obtain the segment address and offset address to the service routine for that interrupt type. The 8259 is initialized to provide the 8088 with interrupt types 08H to 0FH, as illustrated in Table 10-14.

Interrupt Type	Function	Priority
08H	Timer (Channel Zero)	0
09H	Keyboard	1
0AH	Color Adapter Board	2
0BH	N/A	3
0CH	Serial Adapter Board	4
0DH	N/A	5
0EH	Disk Drives	6
0FH	Printer	7

Table 10-7 8259 Interrupt Types

The communications board is assigned to the interrupt vector address 0CH in the interrupt vector table in low memory (see Chapter 3, Figure 3-8). The interrupt type is determined by the values BIOS uses during system initialization. You should not alter these vector types by reprogramming the 8259.

The two registers of interest in the 8259 are the command register and the interrupt mask register. The command register accepts either initialization command words or operational command words. Since BIOS initializes the 8259, I won't get into the many initialization options available. The information is adequately covered in the *Intel Microsystem Components Handbook, Volume I* (#230843-001, pages 2-120 to 2-137), should you want to delve into the initialization mode further.

The only operational command word of interest to us here is the End of Interrupt (EOI) command 20H. Actually, it is referred to in the Intel literature as a specific EOI command. The EOI command must be output to the command register of the 8259 after an interrupt has been serviced. It is the last action taken by the service routine prior to executing the IRET instruction. IRET returns control to the program or task that was executed prior to the interrupt.

The interrupt mask register allows you to mask, or prevent, any of the interrupt types from being generated (disabled). If the bit corresponding to the interrupt type (IRQ0,etc.) is set in the IMR, the interrupt type is disabled. To enable an interrupt type, the corresponding bit in the IMR must be reset. This register can be read (via the IN AL,21H) to determine the currently enabled interrupt types.

MS-DOS RS-232C Functions

Although the functions MS-DOS provides for communications prove to be inadequate except for the simplest of programming chores, I will briefly discuss each of them. The functions can be found in the MACFLE.MAC listing discussed in Chapter 6.

Receive Function

Function: **03H (INT 21H)**

Macro: **@WAITAUX** (See Chapter 6)

This function can be used to retrieve a character from the RHR of the 8250 UART. You should first test bit 0 in the line status register before executing this function. If b0 is set, then a character has been received and is available in the receiver holding register. If you do not check b0 in the line status register, this function will wait until a character is ready or until a BIOS-defined time limit has elapsed (time-out).

This call is relatively useless in a multitasking environment where you cannot wait for anything. Use it if you must, and only when interrupts or other processing chores will not suffer.

To invoke the function, move 03H into AH and execute INT 21H. The character will be returned in AL. If a Control-Z is received, the function will execute an INT 23H which is a Control-C interrupt.

Transmitting Characters

Function: **04H (INT 21H)**

Macro: **@AUXOUT** (See Chapter 6)

This function transmits the character in the DL register to the communications channel. It waits until CTS and DSR are asserted by the other computing device or the DCE (modem). If the signals are not active, the function waits a specified amount of time, then executes a time-out and return without having transmitted the character. You can study the BIOS routine in the *IBM Technical Reference Manual* to understand how this function operates.

BIOS Communication Functions

INT 14H

BIOS offers more than the two functions MS-DOS does. However, BIOS still waits for the electrical signals CTS and DSR to be asserted at the RS-232C interface connector. BIOS adds two other functions, one of which can be used to set the communication parameters of baud rate, word length, parity, and the number of stop bits. The other function can be used to read the line control status and modem status registers. The DX register must contain the serial port desired, zero for the primary port and 1 for the secondary port (the second ACA board, if installed).

Transmitting a Character via BIOS

BIOS Function: **01H (INT 14H)**
[DX] = Serial port number

Macro: **@RS_SEND**

This function sends the character in the AL register to the communications channel. The function waits for the DSR and CTS signals to be asserted prior to sending the character. If bit 7 is set on return from the function, an error occurred and is defined as follows:

```
AH = Error code;

b0 = 1 = RHR full (data ready)

b1 = 1 = Overrun error

b2 = 1 = Parity error

b3 = 1 = Framing error

b4 = 1 = Break detect

b5 = 1 = THR empty

b6 = 1 = Transmitter shift register empty
```

Again, if your application can afford to wait for the signals DSR and CTS to be asserted by the DCE, use this function. If your application does not care about nor require these signals to be present, do not use this function. I'll show you how to transmit a character directly to the port in the programming example presented later in this chapter.

Receiving a Character via BIOS

Function: **02H (INT 14H)**
[DX] = Serial port number

Macro: **@RS_INPUT** (See Chapter 6)

This function waits for a character to arrive at the serial port. It waits for DSR to be asserted before attempting to input the character from the receiver holding register of the UART.

If the signal is not asserted, the routine returns after a specified amount of time with bit 7 in AH set to indicate the time-out. In fact, any nonzero value in AH indicates an error has occurred. The error is mapped in an identical manner as for the transmit function (01H) previously discussed. If AH is zero on returning from this function, the character is in AL.

Setting the Communications Parameters

Function: **00H (INT 14H)**
[DX] = Serial port number

Macro: **@RS232_INIT** (See Chapter 6)

This function sets the baud rate, word length, parity, and number of stop bits required for communications. The bit pattern must be set as shown in Listing 10-1. Once you have decided on the character protocol, move the value into AL, zero into AH, set DX to the serial port number, and execute INT 14H.

Reading the Line and Modem Status

Function: **03H (INT 14H)**
[DX] = Serial port number

Macro: **@RS_STATUS** (See Chapter 6)

This function can be used to obtain the line status and the modem status. The line status is returned in AH and the modem status in AL. The exact bit mapping and significance is illustrated in Listing 10-1.

A Communications Program: COMM.ASM and DLOAD.ASM

In the program COMM.ASM, I have tried to tie all the programming tips and tricks discussed thus far into a real-life application, a communications program. With an assembled version of the programs in Listings 10-1 and 10-2 and a modem, you will be able to communicate with other computers over the telephone.

The program supports a feature known as data capture, whereby all characters received and transmitted are spooled to disk. You can print this file later when you are off-line, saving time and money. If you must obtain an immediate hard copy of the communications in progress, you can also route all received and transmitted data to the printer.

Additionally, the program allows you to transmit an ASCII file via the communications port. This is commonly referred to as downloading a file to another computer. If you own one of the many lap-top computers, such as the Radio Shack Model 100 or 200 or the NEC 8201A, you can use this program to transfer ASCII files from the lap-top computer to the IBM.

The program also illustrates many new concepts, such as circular and double buffering and the installation and support of RS-232 interrupts.

The Main Program Loop

Although the program cannot be considered to be a true multitasking system where multiple jobs or tasks are carried out simultaneously, it gives the appearance that the keyboard disk, communications, printer, and display functions are executing concurrently.

The main loop of the program does nothing except call each functional module within COMM.ASM. The program label MAIN_LOOP to EXIT is the loop that is continually executed. Within the loop, the program checks to see if there is a nonzero value in RAM location EXIT_FLAG. This RAM location contains a nonzero value when the keyboard routine detects an ALT - Z key combination entered from the keyboard. This control is used to terminate the program.

Four basic routines are called from the main program loop: RECEIVE, KEYBOARD, TX_RS232, and PRINTER_OUT_1. I'll discuss each in detail in a moment.

The main program first calls a procedure to initialize the RS-232 and set other functioning parameters of the program. The called procedure, INIT, prompts you for the various parameters and initializes the communications port. This procedure also installs the RS-232 interrupt vector.

Procedure:
INIT

This procedure prompts you for the baud rate desired, the parity type if any, the number of stop bits, and the word length to be used in communications. Each prompting session sets or resets the appropriate bit positions in the RAM location CONFIGURATION. This byte in RAM specifies the character protocol to be used.

Once the parameters have been specified, the program asks you if you want transmitted characters to be echoed to the display. If the communications channel is half duplex, answer yes to the echo prompt. If the channel is full duplex (where the host computer echoes back all characters received from your terminal), answer no to the prompt. Bit 6 in the RAM location SYSTEM_STATUS is set if you answer yes to the prompt and reset if local echo is not desired.

Now find the program label SET_CONFIGURATION in the INIT procedure. The macro @RS232_INIT is invoked, and the parameter CONFIGURATION is passed to the macro. The macro uses the contents of this RAM location to set the character protocol. The macro invokes the BIOS function 00H using BIOS interrupt type 14H. I could have gone directly to the 8250 ports to initialize the character protocol, but this particular BIOS function is easier to use and program than direct port addressing would be.

Installing a New Interrupt Service Routine

Once the 8250 has been initialized with the character protocol, a new interrupt vector that points to the program's RS-232C interrupt service routine is installed. The ES and BX registers are saved, and the MS-DOS function 35H is invoked to read the current interrupt vector.

The macro @READ_VECTOR invokes the MS-DOS function 35H. The value 0CH is passed to the macro. This is the interrupt type for the communications card. When the macro is expanded (see the MACFLE.MAC listing in Appendix D), it moves the interrupt type into AL and the MS-DOS function number 35H into AL and executes the INT 21H instruction. On return, the current interrupt vector for type 0CH will be in the following registers:

```
ES = Segment address of vector.

BX = Offset pointing to the first byte of the service routine.
```

These values should be saved when altering any interrupt vector in low RAM. Prior to terminating your program, the original values should be reinstated in the vector table. In the program, the values are saved in the data segment at RAM locations OLD_VECTOR (segment) and OLD_VECTOR + [2] (offset).

The contents of DS are saved on the stack and the effective address of the new interrupt service routine is moved into the DX register (LEA DX,RS232_INT_1).

The code segment where the routine resides is moved into DS:

```
MOV   AX,MY_CODE

MOV   DS,AX
```

The interrupt type is moved into AL, and the MS-DOS function code 25H is moved into AH. Then the INT 21H instruction is executed. The program then restores DS (POP DS).

Interrupts are disabled (CLI) until the 8250 UART and the 8259 interrupt controller have been initialized. Since you are programming the interrupt control registers of the 8250 UART and the 8259 interrupt controller, it is best to disable all interrupts until your bit twiddling is completed.

RTS and DTR are set to true (space) in the modem status register of the UART. Next, any pending interrupts are reset by performing a dummy read to the line status register. Now it's time to enable the 8250 interrupts. A value of 01H is moved into the AL register. This defines the interrupt mask for the 8250. Since b0 is the only bit set in the mask, only interrupts received by the 8250 are enabled. Anytime the 8250 completely receives a character, an interrupt is generated.

This value is output to the interrupt enable register of the 8250 (port 03F9H). It would be nice and easy if this were all there was to enable interrupts from the ACA board; however, there is one more chip that must be initialized. The 8259 must be able to recognize not just the ACA board interrupts, but it must also recognize any other system peripheral interrupts necessary for the program and the operating system to function properly. In this example, I enabled interrupts for the disk drive, communications board, keyboard, and the 8253 timer. If you disable any of these interrupts, the associated device will not be able to generate an interrupt request to the 8088, and the system will either hang or return erroneous results.

Once you have established the interrupt vector and enabled the interrupt types, the instruction STI is executed, which enables the 8088 to respond to any of the interrupts specified.

Program Operation

You may have come to think of program execution as being carried out one instruction at a time. While it is true that this view of program execution is technically correct, you do have to modify your thinking when working in an interrupt driven environment. Since interrupts can occur at any time from a number of different sources, care must be taken in defining how the program deals with them.

When communications over a high speed channel (9600 baud or greater), a First In, First Out (FIFO) buffer is used to store the characters as they are received via the ACA board. While the ACA receiver interrupt is being serviced, other programming tasks are suspended. Therefore, most all of your interrupt service routines should be kept as short as possible. Leave any processing of the data to other routines which execute during normal program operation, not at interrupt time.

Interrupt Service Routine

The different procedures found in COMM.ASM are assigned very specific functions. The interrupt service routine for the RS-232C must detect any errors in the receiver, such as framing errors, parity errors, overrun errors, or detecting a break condition. The service routine places the ASCII code 1AH in the receive buffer if an error is detected. If no error is detected, the character is input from the UART RHR and placed in the next empty position in the receive buffer.

Notice in the interrupt handler at the program label GET_CHAR, that the DS and ES registers are reintialized to point to the data segment. This is important. BIOS routines (such as disk and timer routines) reinitializes DS and ES. The routines save the registers' contents and point them to scratch areas used by MS-DOS and BIOS. Should an RS-232 interrupt occur when the registers are pointing to a BIOS or MS-DOS data segment, you stand a good chance of wiping out variables used by the operating system. Furthermore, data received during an interrupt will never be saved in the receiver buffer, as the registers are not pointing to the proper data segment.

The interrupt handler adjusts the input pointer RS_IN_POINT and increments the character counter RS_CHARS. The input pointer points to the next open position in the receive buffer. Similarly, there is an output pointer associated with the buffer that points to the next character to be removed from the buffer for processing.

The rule here is: Never assume anything during interrupts. Save the registers on entry, reinitialize them with the data segment of your choice, and restore the DS and ES registers to their former values before executing the IRET instruction.

More on Data Buffers

A buffer is a dedicated portion of contiguous memory used to temporarily hold data. By buffering the data, the devices that the data are destined for have enough time to process the information. There always exists the possibility the devices may be too slow to keep up with the rate at which the buffers are filled. With proper definition at design time, the dynamics of the system can be anticipated and enough time allocated to properly handle data processing.

The buffers used in COMM.ASM are first in, first out (FIFO) buffers. Each buffer has two pointers and a character counter associated with it. One pointer is used as an input pointer, which points to the next available location in the buffer. The other pointer is the output pointer. It points to the next active character position in the buffer. The counter is used in order to avoid lengthy computations to determine if there are characters in the buffer awaiting processing. A routine which must process active characters simply has to test the character count of the buffer. If the value is zero, there are no active characters awaiting processing, and the next task is allowed to execute.

A task which must put characters into the buffer takes the following action:

1. Fetch the buffer's input pointer address.
2. Store the desired value at the byte pointed to by the buffer's input address pointer.
3. Increment the pointer by either 1 or 2, depending on whether a byte or a word had been stored in the buffer.
4. Maintain pointer integrity by making sure the pointer continues to point to the next available location in the buffer.
5. Increment the character counter by one, thus indicating a character has been added to the buffer and awaits processing.

To maintain pointer integrity as mentioned in step four, the address pointer usually has certain bits reset by the routine manipulating the pointer. For example, if the maximum buffer size is 16 bytes, bits 4–15 must always be reset before the routine saves the new value of the offset. This insures the buffer will not extend beyond its allocated memory depth. To demonstrate further, assume we are using a 16-byte buffer and the current offset contains the binary value 0000 0000 0000 1111. The buffer offset is pointing to the last physical byte allocated to the buffer. After incrementing the offset, its new value will be:

```
0000  0000  0001  0000 (binary);
```

which, when added to the virtual start address of the buffer, would be pointing one byte past the end of the buffer.

By resetting bit 4 of the offset value, the offset is reset to 0000 0000 0000 0000, or the first physical location of the buffer. This keeps the buffer circular. When the last physical position of the buffer is occupied, it wraps around and points to the first position. Figure 10-14 illustrates the concept of circular buffering further.

Another important concept is that of double buffering. By providing two or more buffers from which to process characters or data, asynchronous tasks are able to be synchronized to available resources within the system. Each task is assigned specific responsibilities as to when and how data are deposited into a particular buffer and when and how the data are removed from a buffer. Often the data are routed to

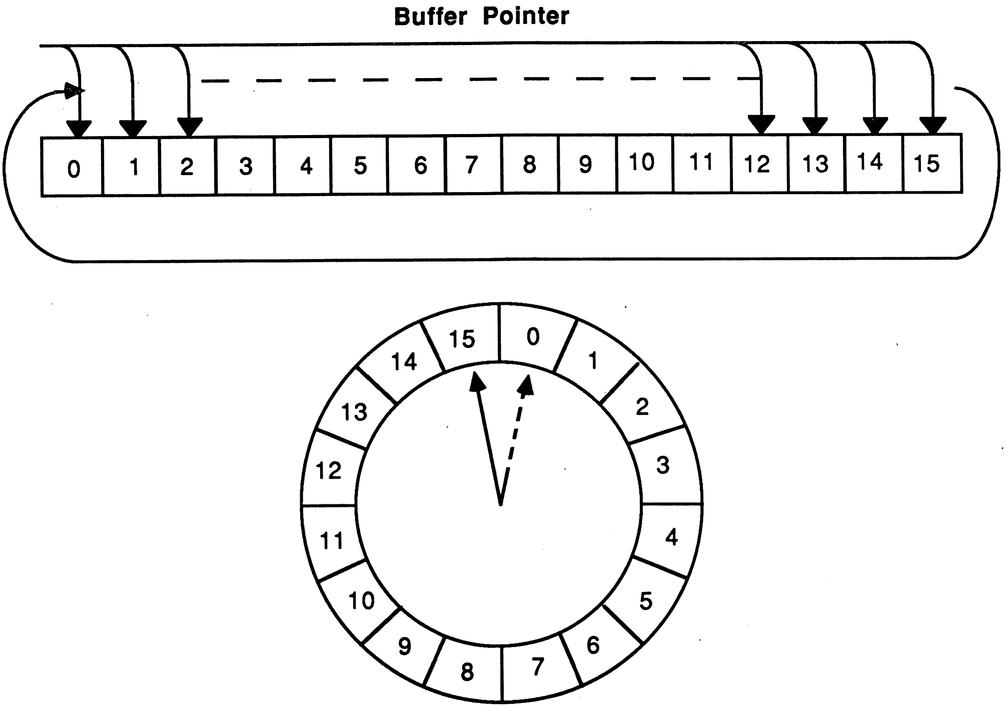


Figure 10-14

another buffer associated with a logical or physical system resource. The second buffer can then be emptied by the task which is associated with that buffer.

Refer to Figure 10-15. Notice the UART receive buffer receives data in a random and asynchronous manner. The arrival of the data is unpredictable as it is received from the communications channel, which is asynchronous. COMM.ASM uses interrupt to alert the processor when a character is received via the RS-232C interface. At interrupt time, the character is put into a buffer associated with the UART's receiver function. Refer now to program Listing 10-1 in Appendix D. In the procedure RS232_INT the interrupt handler is defined.

When a character is put into a buffer, the input pointer to the buffer and the buffer's character count are incremented. When a character is taken from the buffer, the output pointer is incremented to point to the next character's position, and the character count is decremented by one. A procedure can simply check the character count to determine if there are any characters awaiting processing in the buffer.

Find the GET_CHAR label in the program. The input pointer to the receiver buffer is moved to the SI register. After inputting a character from the RHR, the procedure branches to the label POINT_ADJ. The pointer is incremented and logically ANDed with the value 07FFH. Since the buffer for the communications channel is 2048

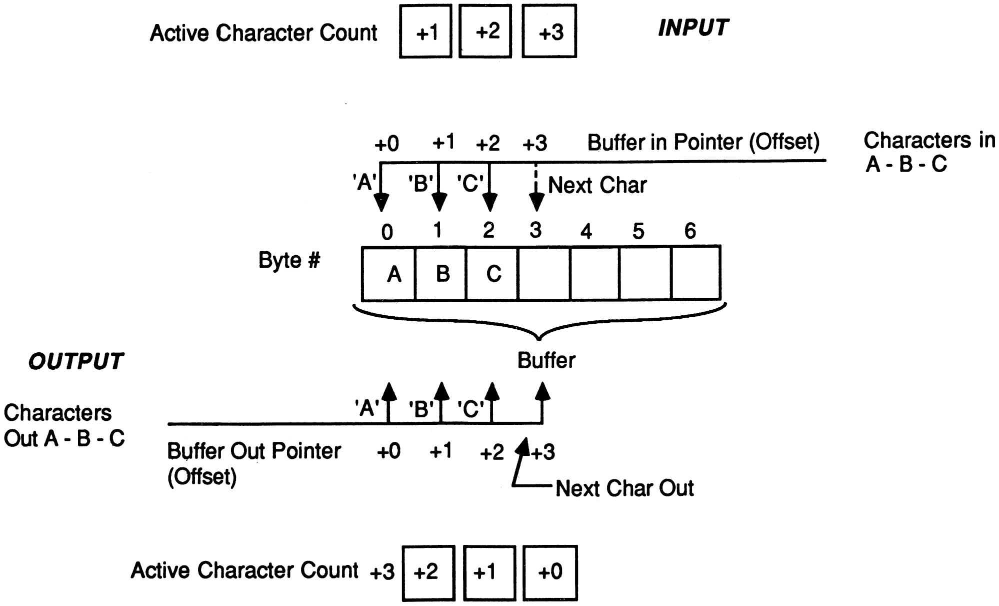

Active Character Count
+1
+2
+3
INPUT
+0 +1 +2 +3 Buffer in Pointer (Offset)
Characters in A - B - C
'A' 'B' 'C'
Next Char
0 1 2 3 4 5 6
Byte #
A B C
Buffer
OUTPUT
Characters Out A - B - C
'A' 'B' 'C'
Buffer Out Pointer (Offset)
+0 +1 +2 +3
Next Char Out
Active Character Count +3
+2
+1
+0

Figure 10-15

bytes in length, this action insures that the pointer will wrap around and point to the beginning of the buffer after the last buffer position has been used.

Nothing more is done with respect to processing the character at this time. The interrupt routine terminates and returns control to the task which was executing when the interrupt occurred.

Procedure: RECEIVE

To demonstrate the concept of double buffering refer to the program listing at the label RECEIVE. This is where the characters previously stored in the UART buffer are processed. Notice that the routine immediately checks the value of RS_CHARS. If the value is zero, there are no characters in the UART buffer, and the routine returns to the main program loop. If there are characters waiting in the buffer, the offset RS_OUT_POINT is added to the buffer's start address RS_BUFFER to obtain the correct pointer value of the next character to process.

The RS_IN_POINT and the RS_OUT_POINT pointers do not have to keep pace with each other. If for some reason the input pointer gets far enough ahead of the output pointer, it is understood that each process has the ability to signal a possible overflow condition and request that other routines (or even the communications channel) be prevented from sending characters to it until it is emptied. This flow control further synchronizes the otherwise asynchronous execution of tasks within the system.

Once a character is received from the buffer, it is processed. The processing occurs at task time. Therefore, it is possible to be buffering additional characters in the primary buffer through interrupts while processing or putting a character into secondary buffers at task time. The character may be routed to the video display, a printer buffer, the capture buffer, or to a disk buffer, if the task associated with the secondary buffer is active. If the task is not active, the character is not put into the task's buffer (see Figure 10-16).

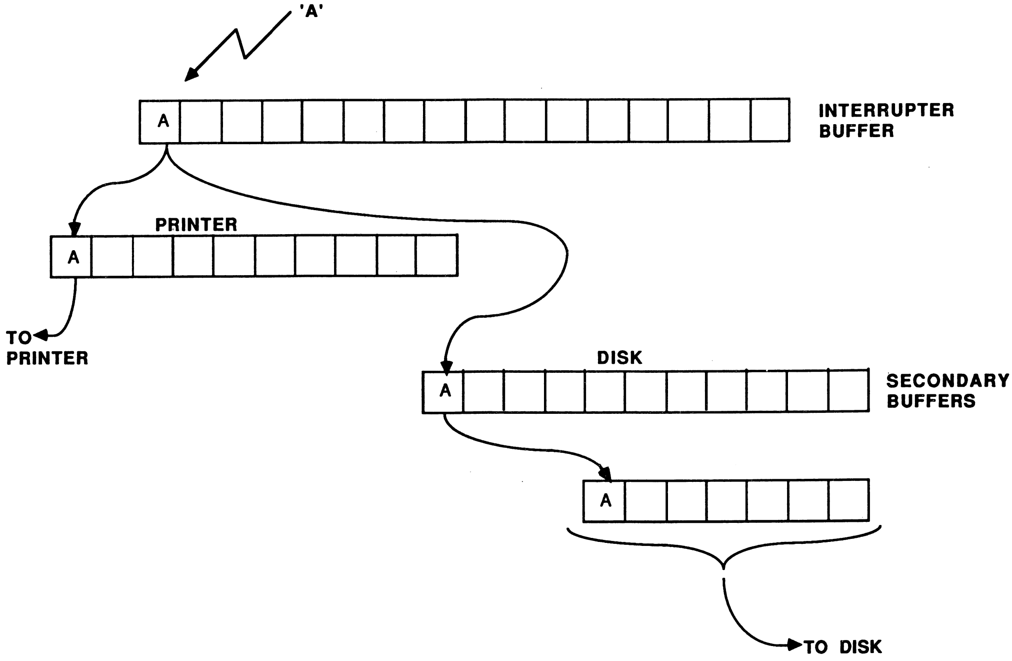


Figure 10-16

Through the use of these secondary buffers, the associated tasks, when called from the job list, must handle the processing of the character. The printer job then gets a character from its buffer and sends it to the printer when and if the printer is ready. The disk spooler uses yet another buffer from which the operating system (MS-DOS) writes a 512-byte sector to disk when the buffer becomes full. It is the responsibility of the lower level tasks to communicate flow control information to the higher level tasks. If secondary buffers are filling up faster than they can be emptied, communications are paced to allow the buffer to be emptied without losing data.

All buffer management, whether at primary or secondary levels, is handled in the same manner as shown in these examples.

Procedure: KEYBOARD

Characters typed from the keyboard are processed by the procedure KEYBOARD. The keyboard routine uses the MS-DOS function 06H to return a character from the keyboard if one is available. DL is set to 0FFH, which informs the function not to echo characters typed to the screen.

The routine checks for the following control characters: DC1, DC2, DC3, and DC4 (ASCII codes 11H, 12H, 13H, and 14H). These controls perform specific functions in the program as illustrated in Figure 10-17.

Keyboard Control	Function
ALT - B	Download an ASCII file
ALT - D	Toggle Disk Spooler On/Off
ALT - H	Display the help menu
ALT - Z	Exit to DOS
CTRL - Q	DC1 (XON)
CTRL - R	DC2 (Printer On)
CTRL - S	DC3 (XOFF)
CTRL - T	DC4 (Printer Off)

Figure 10-17 Control Codes used in COMM.ASM

The keyboard handler must test to see if the ALT key is being pressed in conjunction with another key. Any other character is placed in the transmit buffer at the program label STUFF_BUFFER. The buffer pointer is incremented and kept circular by ANDing the input pointer value with 00FFH. The characters put into the transmit buffer by the keyboard routine are removed from the buffer and transmitted by the procedure TX_RS232.

Procedure: TX_RS232

This procedure is called from the main program loop. The first action taken in this procedure is to ascertain if any characters have been placed in the transmit buffer by the KEYBOARD or DLOAD procedure. If the value of TX_CHARS is zero, there are no characters in the buffer and the procedure returns to the main program loop. If TX_CHARS contains a nonzero value, then the procedure continues.

When the receive buffer stands a chance of overflowing, as when more characters are being received than are being removed, the program or terminal transmits a DC3 (ASCII 13H) to the remote system. It's the computer's way of saying stop sending for a moment. When the situation alleviates itself and the buffer has been adjusted so that the possibility of an overflow has been eliminated or reduced, the program sends a DC1 control (ASCII 11H) to the remote to resume transmission.

This form of flow control, or buffer management, is character oriented as opposed to the electrical pacing techniques that BIOS provides when using DSR with DTR and CTS with RTS in transmission.

The program suspends transmissions when the printer buffer is about to overflow, when the DC3 control is typed from the keyboard (Control-S), or when the remote system sends a DC3 to you. In an ideal situation, the remote terminal would recognize the DC3 as soon as it arrives. In reality, the remote system will have already sent a number of characters before the DC3 is received. Therefore, be sure to allow enough room in the buffer to account for the characters that may still be on the way to your terminal after you have sent the DC3. As a rule of thumb, when a buffer fills to about three-quarters of its capacity, send a DC3, and when the buffer is about one-quarter full, send a DC1 to resume transmissions.

The transmit procedure also returns to the calling program without having sent a character if it finds that the THR of the UART is not empty. You cannot load a character into the THR unless the register is empty.

If there are characters in the TX buffer, no DC3 has been sent and the transmitter holding register of the UART is empty, the next character to be removed from the TX buffer is fetched and transmitted to the communications channel. If the character is a DC3 or a DC1, bit 5 of RAM location SYSTEM_STATUS is set or reset to reflect the control was encountered.

SYSTEM_STATUS is tested to see if bit 6 is set or reset. If the bit is set, then the user elected during the initialization portion of the program to echo all characters transmitted to the display. If the bit is reset, then the procedure will not echo the characters transmitted to the display.

If the character transmitted is carriage return (0DH), then the RAM location SYSTEM_STATUS is checked to see if bit 6 is set. This bit is set during initialization if you have elected to execute a carriage return/linefeed combination when a carriage return is encountered; otherwise, the procedure executes the carriage return only.

Spooling

Procedure: TURN_DISK_ON_OFF

If you type an ALT-D from the keyboard, disk spooling is enabled. When ALT-D is typed once the feature has been enabled, the spool file is closed and spooling to the disk file will cease. ALT-D acts as a toggle to enable or disable the spooling feature.

When invoked, you are prompted for a standard MS-DOS filename, including a drive specification, pathname, filename, and an extension, if desired.

The procedure uses MS-DOS 2.0 disk access mechanisms to create the disk file and write to disk. The MS-DOS function 3CH is used to create the file. The function returns a 16-bit file handle, which must be used during subsequent file access. The file handle is stored in the RAM location FILE_HANDLE in the data segment. The DTA is set to point to DISK_BUFFER in the data segment. If an error occurs during the creation of the file, an error message is displayed and the procedure is terminated.

Procedure: DISK_IN

The procedure DISK_IN stores the character in DL in the DTA of an open file. After each character is transferred to the buffer, the character counter DISK_IN is incremented. When the count equals 512 bytes, a sector is written to disk. If there is a disk error, the file is closed and an error message is displayed. The only control characters that are stored to disk are carriage returns, linefeeds, and the backspace character. All other control characters are ignored by this procedure.

Procedure: PRINTER_IN_1

This procedure is responsible for placing the character in DL into the printer buffer when the printer spooling option is enabled. To enable this option, press Control-R (DC2). To disable printer spooling, press Control-T (DC4). If the buffer fills to 200 characters, the routine automatically sends a DC3 over the communications channel. The PRINTER_OUT_1 procedure is responsible for sending a DC1 to the remote system when the buffer contains fewer than 100 characters.

Procedure: PRINTER_OUT_1

This procedure transmits a character stored in the printer buffer to the printer if one is available. If you toggle the printer on, be sure you have one attached to the system and that it is on-line. Nowhere in the routine is a check made to see if a printer is really part of the system. The program will hang up once the printer buffer exceeds 200 characters if a printer is not attached to the system.

The procedure checks to see if a DC3 printer condition exists and, if so, checks the depth of the printer buffer. If there are fewer than 100 characters in the buffer, a DC1 is transmitted over the communications line, and the DC3 condition is reset. Normal transmission and reception can then resume.

Downloading Files (ALT-B)

When you press ALT-B, the procedure DLOAD is activated (see Listing 10-2 in Appendix D). This file reads a disk file one sector at a time and places the characters read into the transmit buffer. The procedure checks the keyboard during execution for a key closure. If a key has been pressed, the procedure aborts and returns to the terminal mode.

When the download function is first activated, you are prompted for the drive, pathname, filename, and extension of the file to be downloaded. If there is an error in opening the file, the procedure displays an error message and returns to the terminal mode.

The specified file must be an ASCII text file. It cannot be a binary file. Since the program uses XON and XOFF flow control, you cannot transmit a file containing either of these control characters.

Help Menu

Should you forget what the limited control key sequences are which control the program, press ALT-H and a help menu will be displayed. The keyboard procedure in Listing 10-1 contains the routine HELP which clears the screen and displays the help menu.

Exiting the Program

To end the communications program press ALT-Z. This sets the exit flag, which is checked in the main program loop (MAIN_LOOP). When the program finds the exit flag set, the original interrupt vector for communications is restored, and you are returned to MS-DOS.

In Conclusion

You should study the listing of each procedure in detail in Listings 10-1 and 10-2. Experiment and alter the routines. Add new procedures that will increase the utility of the program.

Try adding these features to the program:

1. Character filters to create your own code translation tables for keyboard and received characters.
2. A procedure to transmit a binary file.

The list could go on. I suggest you take a look at some of the commercial programs available and incorporate similar features into your own custom telecommunications program. Assembly Language programming can be rewarding, money saving (you won't have to buy as many programs), and educational. I hope you found the programming examples in this book to be all of these and more.

There were many more programming examples and programming applications I wanted to cover; yet time and space would not permit. The more advanced subjects will have to wait for the follow-up book: *8088/IBM PC Assembly Language Programming—Book II.*

Chapter 10 Review

1. A ____________ is a device that performs parallel to serial conversion of data.
2. Hardware interrupts are prioritized by the ____________ PIC.
3. Full-duplex allows simultaneous ____________ ____________ communications.
4. Simplex is a simple-minded form of conversation. True or False.
5. Asynchronous communication does/does not require a timing element to be transmitted for synchronization.
6. Using PSK modulation, what is the modulation rate of a 1200 bit per second communication channel?
7. The UART found on the serial I/O board is a National Semiconductor ____________ (part number).
8. The line control register of the UART controls the RS-232C signals of the communications line. True or False.

9. Define the 8-bit binary pattern which must be written to the 8259 IMR register to enable only timer, printer, and disk interrupts.

10. The MS-DOS function ___________ can be used to transmit a character via the serial communications adapter.

11. The function listed in question 10 will ___________ until ___________ and ___________ are asserted prior to transmitting a character.

Appendix A

ASCII, BAUDOT, EBCDIC

ASCII is a 7-bit code defining character and control codes from 00H to 7FH.

Baudot is a 5-bit code that defines figures and letters from 00H to 1FH. By use of figures shift or letters shift, the number of characters represented is extended to 60 characters.

EBCDIC is an 8-bit character code that defines characters and control codes from 00H to FFH. It is most often used by IBM mainframes in communications.

Binary	ASCII	BAUDOT Figures/Letters		EBCDIC	Hexidecimal
0000 0000	NULL	Blank	Blank	NULL	00H
0000 0001	SOH	3	E	SOH	01H
0000 0010	STX	Linefeed	Linefeed	STX	02H
0000 0011	ETX	–	A	ETX	03H
0000 0100	EOT	Space	Space	PF	04H
0000 0101	ENQ	,	S	HT	05H
0000 0110	ACK	8	I	LC	06H
0000 0111	BEL	7	U	DEL	07H
0000 1000	BS	CR	CR		08H

Binary	ASCII	BAUDOT Figures/Letters		EBCDIC	Hexidecimal
0000 1001	HT	ENQ	D		09H
0000 1010	LF	4	R	SMM	0AH
0000 1011	VT	BEL	J	VT	0BH
0000 1100	FF	,	N	FF	0CH
0000 1101	CR	$	F	CR	0DH
0000 1110	SO	:	C	SO	0EH
0000 1111	SI	(	K	SI	0FH
0001 0000	DLE	5	T	DLE	10H
0001 0001	DC1	"	Z	DC1	11H
0001 0010	DC2	)	L	DC2	12H
0001 0011	DC3	2	W	TM	13H
0001 0100	DC4	#	H	RES	14H
0001 0101	NAK	6	Y	NL	15H
0001 0110	SYN	0	P	BS	16H
0001 0111	ETB	1	Q	IL	17H
0001 1000	CAN	9	O	CAN	18H
0001 1001	EM	?	B	EM	19H
0001 1010	SUB	&	G	CC	1AH
0001 1011	ESC	[Figures Shift]		CU1	1BH
0001 1100	FS	"	M	IFS	1CH
0001 1101	GS	/	X	IGS	1DH
0001 1110	RS	;	V	IRS	1EH
0001 1111	US	[Letters Shift]		IUS	1FH
0010 0000	SPACE			DS	20H
0010 0001	!			SOS	21H
0010 0010	"			FS	22H
0010 0011	#				23H
0010 0100	$			BYP	24H
0010 0101	%			LF	25H
0010 0110	&			ETB	26H
0010 0111	'			ESC	27H
0010 1000	(				28H
0010 1001	)				29H
0010 1010	*			SM	2AH
0010 1011	+			CU2	2BH
0010 1100	,				2CH
0010 1101	-			ENQ	2DH
0010 1110	.			ACK	2EH
0010 1111	/			BEL	2FH
0011 0000	0				30H
0011 0001	1				31H
0011 0010	2			SYN	32H
0011 0011	3				33H
0011 0100	4			PN	34H
0011 0101	5			RS	35H
0011 0110	6			UC	36H
0011 0111	7			EOT	37H
0011 1000	8				38H
0011 1001	9				39H
0011 1010	:				3AH

Binary	ASCII	BAUDOT Figures/Letters	EBCDIC	Hexidecimal
0011 1011	;		CU3	3BH
0011 1100	<		DC4	3CH
0011 1101	=		NAK	3DH
0011 1110	>			3EH
0011 1111	?		SUB	3FH
0100 0000	@		SPACE	40H
0100 0001	A			41H
0100 0010	B			42H
0100 0011	C			43H
0100 0100	D			44H
0100 0101	E			45H
0100 0110	F			46H
0100 0111	G			47H
0100 1000	H			48H
0100 1001	I			49H
0100 1010	J			4AH
0100 1011	K		.	4BH
0100 1100	L		<	4CH
0100 1101	M		(	4DH
0100 1110	N		+	4EH
0100 1111	O		|	4FH
0101 0000	P		[	50H
0101 0001	Q			51H
0101 0010	R			52H
0101 0011	S			53H
0101 0100	T			54H
0101 0101	U			55H
0101 0110	V			56H
0101 0111	W			57H
0101 1000	X			58H
0101 1001	Y			59H
0101 1010	Z		!	5AH
0101 1011	[		$	5BH
0101 1100	\		*	5CH
0101 1101	]		)	5DH
0101 1110	^		;	5EH
0101 1111	_		~	5FH
0110 0000	\		-	60H
0110 0001	a		/	61H
0110 0010	b			62H
0110 0011	c			63H
0110 0100	d			64H
0110 0101	e			65H
0110 0110	f			66H
0110 0111	g			67H
0110 1000	h			68H
0110 1001	i			69H
0110 1010	j			6AH
0110 1011	k		,	6BH
0110 1100	l		%	6CH

Binary	ASCII	BAUDOT Figures/Letters	EBCDIC	Hexidecimal
0110 1101	m			6DH
0110 1110	n		>	6EH
0110 1111	o		?	6FH
0111 0000	p			70H
0111 0001	q			71H
0111 0010	r			72H
0111 0011	s			73H
0111 0100	t			74H
0111 0101	u			75H
0111 0110	v			76H
0111 0111	w			77H
0111 1000	x			78H
0111 1001	y			79H
0111 1010	z		:	7AH
0111 1011	{		#	7BH
0111 1100	|		@	7CH
0111 1101	}			7DH
0111 1110	∿		=	7EH
0111 1111	RUBOUT (DEL)		"	7FH
1000 0000				80H
1000 0001			a	81H
1000 0010			b	82H
1000 0011			c	83H
1000 0100			d	84H
1000 0101			e	85H
1000 0110			f	86H
1000 0111			g	87H
1000 1000			h	88H
1000 1001			i	89H
..				..
..				..
1001 0001			j	91H
1001 0010			k	92H
1001 0011			l	93H
1001 0100			m	94H
1001 0101			n	95H
1001 0110			o	96H
1001 0111			p	97H
1001 1000			q	98H
1001 1001			r	99H
..				..
..				..
1010 0010			s	A2H
1010 0011			t	A3H
1010 0100			u	A4H
1010 0101			v	A5H
1010 0110			w	A6H
1010 0111			x	A7H
1010 1000			y	A8H

Binary	ASCII	BAUDOT Figures/Letters	EBCDIC	Hexidecimal
1010 1001			z	A9H
..				..
..				..
1100 0000			{	C0H
1100 0001			A	C1H
1100 0010			B	C2H
1100 0011			C	C3H
1100 0100			D	C4H
1100 0101			E	C5H
1100 0110			F	C6H
1100 0111			G	C7H
1100 1000			H	C8H
1100 1001			I	C9H
..				..
..				..
1101 0000			}	D0H
1101 0001			J	D1H
1101 0010			K	D2H
1101 0011			L	D3H
1101 0100			M	D4H
1101 0101			N	D5H
1101 0110			O	D6H
1101 0111			P	D7H
1101 1000			Q	D8H
1101 1001			R	D9H
..				..
..				..
1110 0010			S	E2H
1110 0011			T	E3H
1110 0100			U	E4H
1110 0101			V	E5H
1110 0110			W	E6H
1110 0111			X	E7H
1110 1000			Y	E8H
1110 1001			Z	E9H
..				..
..				..
1111 0000			0	F0H
1111 0001			1	F1H
1111 0010			2	F2H
1111 0011			3	F3H
1111 0100			4	F4H
1111 0101			5	F5H
1111 0110			6	F6H
1111 0111			7	F7H
1111 1000			8	F8H
1111 1001			9	F9H

Appendix B

8088/86 Instruction Set

The following is the complete 8088/86 instruction set as published in the *Intel iAPX 88 Book* (pages 2-51 through 2-68), Intel order number 210200-002. The instruction set is reprinted with the permission of Intel Corporation.

Table 2-20. Effective Address Calculation Time

EA COMPONENTS		CLOCKS*
Displacement Only		6
Base or Index Only	(BX,BP,SI,DI)	5
Displacement + Base or Index	(BX,BP,SI,DI)	9
Base + Index	BP+DI, BX+SI	7
	BP+SI, BX+DI	8
Displacement + Base + Index	BP+DI+DISP BX+SI+DISP	11
	BP+SI+DISP BX+DI+DISP	12

*Add 2 clocks for segment override

that the BIU can obtain the bus on demand, i.e., that no other processors are competing for the bus.)

With typical instruction mixes, the time actually required to execute a sequence of instructions will typically be within 5-10% of the sum of the individual timings given in table 2-21. Cases can be constructed, however, in which execution time may be much higher than the sum of the figures provided in the table. The execution time for a given sequence of instructions, however, is always repeatable, assuming comparable external conditions (interrupts, coprocessor activity, etc.). If the execution time for a given series of instructions must be determined exactly, the instructions should be run on an execution vehicle such as the SDK-86 or the iSBC 86/12™ board.

Table 2-21. Instruction Set Reference Data

AAA	**AAA** (no operands) ASCII adjust for addition

Flags	O	D	I	T	S	Z	A	P	C
	U				U	U	X	U	X

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	4	—	1	AAA

AAD	**AAD** (no operands) ASCII adjust for division

Flags	O	D	I	T	S	Z	A	P	C
	U				X	X	U	X	U

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	60	—	2	AAD

AAM	**AAM** (no operands) ASCII adjust for multiply

Flags	O	D	I	T	S	Z	A	P	C
	U				X	X	U	X	U

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	83	—	1	AAM

AAS	**AAS** (no operands) ASCII adjust for subtraction

Flags	O	D	I	T	S	Z	A	P	C
	U				U	U	X	U	X

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	4	—	1	AAS

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

ADC	**ADC** destination,source Add with carry	**Flags**	O	D	I	T	S	Z	A	P	C
			X				X	X	X	X	X

Operands	Clocks	Transfers*	Bytes	Coding Example
register, register	3	—	2	ADC AX, SI
register, memory	9+EA	1	2-4	ADC DX, BETA [SI]
memory, register	16+EA	2	2-4	ADC ALPHA [BX] [SI], DI
register, immediate	4	—	3-4	ADC BX, 256
memory, immediate	17+EA	2	3-6	ADC GAMMA, 30H
accumulator, immediate	4	—	2-3	ADC AL, 5

ADD	**ADD** destination,source Addition	**Flags**	O	D	I	T	S	Z	A	P	C
			X				X	X	X	X	X

Operands	Clocks	Transfers*	Bytes	Coding Example
register, register	3	—	2	ADD CX, DX
register, memory	9+EA	1	2-4	ADD DI, [BX].ALPHA
memory, register	16+EA	2	2-4	ADD TEMP, CL
register, immediate	4	—	3-4	ADD CL, 2
memory, immediate	17+EA	2	3-6	ADD ALPHA, 2
accumulator, immediate	4	—	2-3	ADD AX, 200

AND	**AND** destination,source Logical and	**Flags**	O	D	I	T	S	Z	A	P	C
			0				X	X	U	X	0

Operands	Clocks	Transfers*	Bytes	Coding Example
register, register	3	—	2	AND AL,BL
register, memory	9+EA	1	2-4	AND CX,FLAG_WORD
memory, register	16+EA	2	2-4	AND ASCII [DI],AL
register, immediate	4	—	3-4	AND CX,0F0H
memory, immediate	17+EA	2	3-6	AND BETA, 01H
accumulator, immediate	4	—	2-3	AND AX, 01010000B

CALL	**CALL** target Call a procedure	**Flags**	O	D	I	T	S	Z	A	P	C

Operands	Clocks	Transfers*	Bytes	Coding Examples
near-proc	19	1	3	CALL NEAR_PROC
far-proc	28	2	5	CALL FAR_PROC
memptr 16	21+EA	2	2-4	CALL PROC_TABLE [SI]
regptr 16	16	1	2	CALL AX
memptr 32	37+EA	4	2-4	CALL [BX].TASK [SI]

CBW	**CBW** (no operands) Convert byte to word	**Flags**	O	D	I	T	S	Z	A	P	C

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	2	—	1	CBW

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Mnemonics © Intel, 1978

Table 2-21. Instruction Set Reference Data (Cont'd.)

CLC

CLC (no operands)
Clear carry flag

Flags	O	D	I	T	S	Z	A	P	C
									0

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	2	—	1	CLC

CLD

CLD (no operands)
Clear direction flag

Flags	O	D	I	T	S	Z	A	P	C
		0							

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	2	—	1	CLD

CLI

CLI (no operands)
Clear interrupt flag

Flags	O	D	I	T	S	Z	A	P	C
			0						

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	2	—	1	CLI

CMC

CMC (no operands)
Complement carry flag

Flags	O	D	I	T	S	Z	A	P	C
									X

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	2	—	1	CMC

CMP

CMP destination,source
Compare destination to source

Flags	O	D	I	T	S	Z	A	P	C
	X				X	X	X	X	X

Operands	Clocks	Transfers*	Bytes	Coding Example
register, register	3	—	2	CMP BX, CX
register, memory	9+EA	1	2-4	CMP DH, ALPHA
memory, register	9+EA	1	2-4	CMP [BP+2], SI
register, immediate	4	—	3-4	CMP BL, 02H
memory, immediate	10+EA	1	3-6	CMP [BX].RADAR [DI], 3420H
accumulator, immediate	4	—	2-3	CMP AL, 00010000B

CMPS

CMPS dest-string,source-string
Compare string

Flags	O	D	I	T	S	Z	A	P	C
	X				X	X	X	X	X

Operands	Clocks	Transfers*	Bytes	Coding Example
dest-string, source-string	22	2	1	CMPS BUFF1, BUFF2
(repeat) dest-string, source-string	9+22/rep	2/rep	1	REPE CMPS ID, KEY

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

		Flags	O	D	I	T	S	Z	A	P	C
CWD	**CWD** (no operands) Convert word to doubleword										

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	5	—	1	CWD

		Flags	O	D	I	T	S	Z	A	P	C
DAA	**DAA** (no operands) Decimal adjust for addition		X				X	X	X	X	X

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	4	—	1	DAA

		Flags	O	D	I	T	S	Z	A	P	C
DAS	**DAS** (no operands) Decimal adjust for subtraction		U				X	X	X	X	X

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	4	—	1	DAS

		Flags	O	D	I	T	S	Z	A	P	C
DEC	**DEC** destination Decrement by 1		X				X	X	X	X	

Operands	Clocks	Transfers*	Bytes	Coding Example
reg16	2	—	1	DEC AX
reg8	3	—	2	DEC AL
memory	15+EA	2	2-4	DEC ARRAY [SI]

		Flags	O	D	I	T	S	Z	A	P	C
DIV	**DIV** source Division, unsigned		U				U	U	U	U	U

Operands	Clocks	Transfers*	Bytes	Coding Example
reg8	80-90	—	2	DIV CL
reg16	144-162	—	2	DIV BX
mem8	(86-96) +EA	1	2-4	DIV ALPHA
mem16	(150-168) +EA	1	2-4	DIV TABLE [SI]

		Flags	O	D	I	T	S	Z	A	P	C
ESC	**ESC** external-opcode,source Escape										

Operands	Clocks	Transfers*	Bytes	Coding Example
immediate, memory	8+EA	1	2-4	ESC 6,ARRAY [SI]
immediate, register	2	—	2	ESC 20,AL

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

HLT	**HLT** (no operands) Halt	**Flags**	O	D	I	T	S	Z	A	P	C

Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	HLT

IDIV	**IDIV** source Integer division	**Flags**	O	D	I	T	S	Z	A	P	C
			U				U	U	U	U	U

Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg8	101-112	—	2	IDIV BL
reg16	165-184	—	2	IDIV CX
mem8	(107-118) +EA	1	2-4	IDIV DIVISOR_BYTE [SI]
mem16	(171-190) +EA	1	2-4	IDIV [BX].DIVISOR_WORD

IMUL	**IMUL** source Integer multiplication	**Flags**	O	D	I	T	S	Z	A	P	C
			X				U	U	U	U	X

Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg8	80-98	—	2	IMUL CL
reg16	128-154	—	2	IMUL BX
mem8	(86-104) +EA	1	2-4	IMUL RATE_BYTE
mem16	(134-160) +EA	1	2-4	IMUL RATE_WORD [BP] [DI]

IN	**IN** accumulator,port Input byte or word	**Flags**	O	D	I	T	S	Z	A	P	C

Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
accumulator, immed8	10	1	2	IN AL, 0FFEAH
accumulator, DX	8	1	1	IN AX, DX

INC	**INC** destination Increment by 1	**Flags**	O	D	I	T	S	Z	A	P	C
			X				X	X	X	X	

Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg16	2	—	1	INC CX
reg8	3	—	2	INC BL
memory	15+EA	2	2-4	INC ALPHA [DI] [BX]

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Mnemonics © Intel, 1978

Table 2-21. Instruction Set Reference Data (Cont'd.)

INT	**INT** interrupt-type Interrupt	**Flags**	O	D	I	T	S	Z	A	P	C
					0	0					

Operands	Clocks	Transfers*	Bytes	Coding Example
immed8 (type = 3)	52	5	1	INT 3
immed8 (type ≠ 3)	51	5	2	INT 67

INTR†	**INTR** (external maskable interrupt) Interrupt if INTR and IF=1	**Flags**	O	D	I	T	S	Z	A	P	C
					0	0					

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	61	7	N/A	N/A

INTO	**INTO** (no operands) Interrupt if overflow	**Flags**	O	D	I	T	S	Z	A	P	C
					0	0					

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	53 or 4	5	1	INTO

IRET	**IRET** (no operands) Interrupt Return	**Flags**	O	D	I	T	S	Z	A	P	C
			R	R	R	R	R	R	R	R	R

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	24	3	1	IRET

JA/JNBE	**JA/JNBE** short-label Jump if above/Jump if not below nor equal	**Flags**	O	D	I	T	S	Z	A	P	C

Operands	Clocks	Transfers*	Bytes	Coding Example
short-label	16 or 4	—	2	JA ABOVE

JAE/JNB	**JAE/JNB** short-label Jump if above or equal/Jump if not below	**Flags**	O	D	I	T	S	Z	A	P	C

Operands	Clocks	Transfers*	Bytes	Coding Example
short-label	16 or 4	—	2	JAE ABOVE_EQUAL

JB/JNAE	**JB/JNAE** short-label Jump if below/Jump if not above nor equal	**Flags**	O	D	I	T	S	Z	A	P	C

Operands	Clocks	Transfers*	Bytes	Coding Example
short-label	16 or 4	—	2	JB BELOW

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

†INTR is not an instruction; it is included in table 2-21 only for timing information.

Table 2-21. Instruction Set Reference Data (Cont'd.)

JBE/JNA

JBE/JNA short-label
Jump if below or equal/Jump if not above

Flags O D I T S Z A P C

Operands	Clocks	Transfers*	Bytes	Coding Example
short-label	16 or 4	—	2	JNA NOT_ABOVE

JC

JC short-label
Jump if carry

Flags O D I T S Z A P C

Operands	Clocks	Transfers*	Bytes	Coding Example
short-label	16 or 4	—	2	JC CARRY_SET

JCXZ

JCXZ short-label
Jump if CX is zero

Flags O D I T S Z A P C

Operands	Clocks	Transfers*	Bytes	Coding Example
short-label	18 or 6	—	2	JCXZ COUNT_DONE

JE/JZ

JE/JZ short-label
Jump if equal/Jump if zero

Flags O D I T S Z A P C

Operands	Clocks	Transfers*	Bytes	Coding Example
short-label	16 or 4	—	2	JZ ZERO

JG/JNLE

JG/JNLE short-label
Jump if greater/Jump if not less nor equal

Flags O D I T S Z A P C

Operands	Clocks	Transfers*	Bytes	Coding Example
short-label	16 or 4	—	2	JG GREATER

JGE/JNL

JGE/JNL short-label
Jump if greater or equal/Jump if not less

Flags O D I T S Z A P C

Operands	Clocks	Transfers*	Bytes	Coding Example
short-label	16 or 4	—	2	JGE GREATER_EQUAL

JL/JNGE

JL/JNGE short-label
Jump if less/Jump if not greater nor equal

Flags O D I T S Z A P C

Operands	Clocks	Transfers*	Bytes	Coding Example
short-label	16 or 4	—	2	JL LESS

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Mnemonics © Intel, 1978

Table 2-21. Instruction Set Reference Data (Cont'd.)

JLE/JNG	**JLE/JNG** short-label Jump if less or equal/Jump if not greater			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNG NOT_GREATER

JMP	**JMP** target Jump			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	15	—	2	JMP SHORT
near-label	15	—	3	JMP WITHIN_SEGMENT
far-label	15	—	5	JMP FAR_LABEL
memptr16	18+EA	1	2-4	JMP [BX].TARGET
regptr16	11	—	2	JMP CX
memptr32	24+EA	2	2-4	JMP OTHER.SEG [SI]

JNC	**JNC** short-label Jump if not carry			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNC NOT_CARRY

JNE/JNZ	**JNE/JNZ** short-label Jump if not equal/Jump if not zero			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNE NOT_EQUAL

JNO	**JNO** short-label Jump if not overflow			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNO NO_OVERFLOW

JNP/JPO	**JNP/JPO** short-label Jump if not parity/Jump if parity odd			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JPO ODD_PARITY

JNS	**JNS** short-label Jump if not sign			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNS POSITIVE

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Mnemonics © Intel, 1978

Table 2-21. Instruction Set Reference Data (Cont'd.)

JO	**JO** short-label Jump if overflow	**Flags** O D I T S Z A P C

Operands	Clocks	Transfers*	Bytes	Coding Example
short-label	16 or 4	—	2	JO SIGNED_OVRFLW

JP/JPE	**JP/JPE** short-label Jump if parity/Jump if parity even	**Flags** O D I T S Z A P C

Operands	Clocks	Transfers*	Bytes	Coding Example
short-label	16 or 4	—	2	JPE EVEN_PARITY

JS	**JS** short-label Jump if sign	**Flags** O D I T S Z A P C

Operands	Clocks	Transfers*	Bytes	Coding Example
short-label	16 or 4	—	2	JS NEGATIVE

LAHF	**LAHF** (no operands) Load AH from flags	**Flags** O D I T S Z A P C

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	4	—	1	LAHF

LDS	**LDS** destination,source Load pointer using DS	**Flags** O D I T S Z A P C

Operands	Clocks	Transfers	Bytes	Coding Example
reg16, mem32	16+EA	2	2-4	LDS SI,DATA.SEG [DI]

LEA	**LEA** destination,source Load effective address	**Flags** O D I T S Z A P C

Operands	Clocks	Transfers*	Bytes	Coding Example
reg16, mem16	2+EA	—	2-4	LEA BX, [BP] [DI]

LES	**LES** destination,source Load pointer using ES	**Flags** O D I T S Z A P C

Operands	Clocks	Transfers*	Bytes	Coding Example
reg16, mem32	16+EA	2	2-4	LES DI, [BX].TEXT_BUFF

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Mnemonics © Intel, 1978

Table 2-21. Instruction Set Reference Data (Cont'd.)

LOCK	**LOCK** (no operands) Lock bus			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	LOCK XCHG FLAG,AL

LODS	**LODS** source-string Load string			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
source-string	12	1	1	LODS CUSTOMER_NAME
(repeat) source-string	9+13/rep	1/rep	1	REP LODS NAME

LOOP	**LOOP** short-label Loop			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	17/5	—	2	LOOP AGAIN

LOOPE/LOOPZ	**LOOPE/LOOPZ** short-label Loop if equal/Loop if zero			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	18 or 6	—	2	LOOPE AGAIN

LOOPNE/LOOPNZ	**LOOPNE/LOOPNZ** short-label Loop if not equal/Loop if not zero			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	19 or 5	—	2	LOOPNE AGAIN

NMI†	**NMI** (external nonmaskable interrupt) Interrupt if NMI = 1			**Flags** O S I T S Z A P C 0 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	50	5	N/A	N/A

---

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

†NMI is not an instruction; it is included in table 2-21 only for timing information.

Table 2-21. Instruction Set Reference Data (Cont'd.)

MOV	**MOV** destination,source Move	**Flags**	O	D	I	T	S	Z	A	P	C

Operands	Clocks	Transfers*	Bytes	Coding Example
memory, accumulator	10	1	3	MOV ARRAY [SI], AL
accumulator, memory	10	1	3	MOV AX, TEMP_RESULT
register, register	2	—	2	MOV AX,CX
register, memory	8+EA	1	2-4	MOV BP, STACK_TOP
memory, register	9+EA	1	2-4	MOV COUNT [DI], CX
register, immediate	4	—	2-3	MOV CL, 2
memory, immediate	10+EA	1	3-6	MOV MASK [BX] [SI], 2CH
seg-reg, reg16	2	—	2	MOV ES, CX
seg-reg, mem16	8+EA	1	2-4	MOV DS, SEGMENT_BASE
reg16, seg-reg	2	—	2	MOV BP, SS
memory, seg-reg	9+EA	1	2-4	MOV [BX].SEG_SAVE, CS

MOVS	**MOVS** dest-string,source-string Move string	**Flags**	O	D	I	T	S	Z	A	P	C

Operands	Clocks	Transfers*	Bytes	Coding Example
dest-string, source-string	18	2	1	MOVS LINE EDIT_DATA
(repeat) dest-string, source-string	9+17/rep	2/rep	1	REP MOVS SCREEN, BUFFER

MOVSB/MOVSW	**MOVSB/MOVSW** (no operands) Move string (byte/word)	**Flags**	O	D	I	T	S	Z	A	P	C

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	18	2	1	MOVSB
(repeat) (no operands)	9+17/rep	2/rep	1	REP MOVSW

MUL	**MUL** source Multiplication, unsigned	**Flags**	O	D	I	T	S	Z	A	P	C
			X				U	U	U	U	X

Operands	Clocks	Transfers*	Bytes	Coding Example
reg8	70-77	—	2	MUL BL
reg16	118-133	—	2	MUL CX
mem8	(76-83) +EA	1	2-4	MUL MONTH [SI]
mem16	(124-139) +EA	1	2-4	MUL BAUD_RATE

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

NEG

NEG destination
Negate

Flags	O	D	I	T	S	Z	A	P	C
	X				X	X	X	X	1*

Operands	Clocks	Transfers*	Bytes	Coding Example
register	3	—	2	NEG AL
memory	16+EA	2	2-4	NEG MULTIPLIER

*0 if destination = 0

NOP

NOP (no operands)
No Operation

Flags	O	D	I	T	S	Z	A	P	C

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	3	—	1	NOP

NOT

NOT destination
Logical not

Flags	O	D	I	T	S	Z	A	P	C

Operands	Clocks	Transfers*	Bytes	Coding Example
register	3	—	2	NOT AX
memory	16+EA	2	2-4	NOT CHARACTER

OR

OR destination,source
Logical inclusive or

Flags	O	D	I	T	S	Z	A	P	C
	0				X	X	U	X	0

Operands	Clocks	Transfers*	Bytes	Coding Example
register, register	3	—	2	OR AL, BL
register, memory	9+EA	1	2-4	OR DX, PORT_ID [DI]
memory, register	16+EA	2	2-4	OR FLAG_BYTE, CL
accumulator, immediate	4	—	2-3	OR AL, 01101100B
register, immediate	4	—	3-4	OR CX,01H
memory, immediate	17+EA	2	3-6	OR [BX].CMD_WORD,0CFH

OUT

OUT port,accumulator
Output byte or word

Flags	O	D	I	T	S	Z	A	P	C

Operands	Clocks	Transfers*	Bytes	Coding Example
immed8, accumulator	10	1	2	OUT 44, AX
DX, accumulator	8	1	1	OUT DX, AL

POP

POP destination
Pop word off stack

Flags	O	D	I	T	S	Z	A	P	C

Operands	Clocks	Transfers*	Bytes	Coding Example
register	8	1	1	POP DX
seg-reg (CS illegal)	8	1	1	POP DS
memory	17+EA	2	2-4	POP PARAMETER

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Mnemonics © Intel, 1978

Table 2-21. Instruction Set Reference Data (Cont'd.)

POPF	**POPF** (no operands) Pop flags off stack

Flags	O	D	I	T	S	Z	A	P	C
	R	R	R	R	R	R	R	R	R

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	8	1	1	POPF

PUSH	**PUSH** source Push word onto stack

Flags	O	D	I	T	S	Z	A	P	C

Operands	Clocks	Transfers*	Bytes	Coding Example
register	11	1	1	PUSH SI
seg-reg (CS legal)	10	1	1	PUSH ES
memory	16+EA	2	2-4	PUSH RETURN_CODE [SI]

PUSHF	**PUSHF** (no operands) Push flags onto stack

Flags	O	D	I	T	S	Z	A	P	C

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	10	1	1	PUSHF

RCL	**RCL** destination,count Rotate left through carry

Flags	O	D	I	T	S	Z	A	P	C
	X								X

Operands	Clocks	Transfers*	Bytes	Coding Example
register, 1	2	—	2	RCL CX, 1
register, CL	8+4/bit	—	2	RCL AL, CL
memory, 1	15+EA	2	2-4	RCL ALPHA, 1
memory, CL	20+EA+ 4/bit	2	2-4	RCL [BP].PARM, CL

RCR	**RCR** designation,count Rotate right through carry

Flags	O	D	I	T	S	Z	A	P	C
	X								X

Operands	Clocks	Transfers*	Bytes	Coding Example
register, 1	2	—	2	RCR BX, 1
register, CL	8+4/bit	—	2	RCR BL, CL
memory, 1	15+EA	2	2-4	RCR [BX].STATUS, 1
memory, CL	20+EA+ 4/bit	2	2-4	RCR ARRAY [DI], CL

REP	**REP** (no operands) Repeat string operation

Flags	O	D	I	T	S	Z	A	P	C

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	2	—	1	REP MOVS DEST, SRCE

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Mnemonics © Intel, 1978

Table 2-21. Instruction Set Reference Data (Cont'd.)

REPE/REPZ	**REPE/REPZ** (no operands) Repeat string operation while equal/while zero	**Flags**	O	D	I	T	S	Z	A	P	C

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	2	—	1	REPE CMPS DATA, KEY

REPNE/REPNZ	**REPNE/REPNZ** (no operands) Repeat string operation while not equal/not zero	**Flags**	O	D	I	T	S	Z	A	P	C

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	2	—	1	REPNE SCAS INPUT_LINE

RET	**RET** optional-pop-value Return from procedure	**Flags**	O	D	I	T	S	Z	A	P	C

Operands	Clocks	Transfers*	Bytes	Coding Example
(intra-segment, no pop)	8	1	1	RET
(intra-segment, pop)	12	1	3	RET 4
(inter-segment, no pop)	18	2	1	RET
(inter-segment, pop)	17	2	3	RET 2

ROL	**ROL** destination,count Rotate left	**Flags**	O	D	I	T	S	Z	A	P	C
			X								X

Operands	Clocks	Transfers	Bytes	Coding Examples
register, 1	2	—	2	ROL BX, 1
register, CL	8 + 4/bit	—	2	ROL DI, CL
memory, 1	15 + EA	2	2-4	ROL FLAG_BYTE [DI],1
memory, CL	20 + EA + 4/bit	2	2-4	ROL ALPHA , CL

ROR	**ROR** destination,count Rotate right	**Flags**	O	D	I	T	S	Z	A	P	C
			X								X

Operand	Clocks	Transfers*	Bytes	Coding Example
register, 1	2	—	2	ROR AL, 1
register, CL	8 + 4/bit	—	2	ROR BX, CL
memory, 1	15 + EA	2	2-4	ROR PORT_STATUS, 1
memory, CL	20 + EA + 4/bit	2	2-4	ROR CMD_WORD, CL

SAHF	**SAHF** (no operands) Store AH into flags	**Flags**	O	D	I	T	S	Z	A	P	C
							R	R	R	R	R

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	4	—	1	SAHF

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

SAL/SHL

SAL/SHL destination,count
Shift arithmetic left/Shift logical left

Flags	O	D	I	T	S	Z	A	P	C
	X								X

Operands	Clocks	Transfers*	Bytes	Coding Examples
register,1	2	—	2	SAL AL,1
register, CL	8+4/bit	—	2	SHL DI, CL
memory,1	15+EA	2	2-4	SHL [BX].OVERDRAW, 1
memory, CL	20+EA+4/bit	2	2-4	SAL STORE_COUNT, CL

SAR

SAR destination,source
Shift arithmetic right

Flags	O	D	I	T	S	Z	A	P	C
	X				X	X	U	X	X

Operands	Clocks	Transfers*	Bytes	Coding Example
register, 1	2	—	2	SAR DX, 1
register, CL	8+4/bit	—	2	SAR DI, CL
memory, 1	15+EA	2	2-4	SAR N_BLOCKS, 1
memory, CL	20+EA+4/bit	2	2-4	SAR N_BLOCKS, CL

SBB

SBB destination,source
Subtract with borrow

Flags	O	D	I	T	S	Z	A	P	C
	X				X	X	X	X	X

Operands	Clocks	Transfers*	Bytes	Coding Example
register, register	3	—	2	SBB BX, CX
register, memory	9+EA	1	2-4	SBB DI, [BX].PAYMENT
memory, register	16+EA	2	2-4	SBB BALANCE, AX
accumulator, immediate	4	—	2-3	SBB AX, 2
register, immediate	4	—	3-4	SBB CL, 1
memory, immediate	17+EA	2	3-6	SBB COUNT [SI], 10

SCAS

SCAS dest-string
Scan string

Flags	O	D	I	T	S	Z	A	P	C
	X				X	X	X	X	X

Operands	Clocks	Transfers*	Bytes	Coding Example
dest-string	15	1	1	SCAS INPUT_LINE
(repeat) dest-string	9+15/rep	1/rep	1	REPNE SCAS BUFFER

SEGMENT†

SEGMENT override prefix
Override to specified segment

Flags	O	D	I	T	S	Z	A	P	C

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	2	—	1	MOV SS:PARAMETER, AX

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

†ASM-86 incorporates the segment override prefix into the operand specification and not as a separate instruction. SEGMENT is included in table 2-21 only for timing information.

Mnemonics © Intel, 1978

Table 2-21. Instruction Set Reference Data (Cont'd.)

SHR	**SHR** destination,count Shift logical right	Flags	O	D	I	T	S	Z	A	P	C
			X								X

Operands	Clocks	Transfers*	Bytes	Coding Example
register, 1	2	—	2	SHR SI, 1
register, CL	8+4/bit	—	2	SHR SI, CL
memory, 1	15+EA	2	2-4	SHR ID_BYTE [SI] [BX], 1
memory, CL	20+EA+ 4/bit	2	2-4	SHR INPUT_WORD, CL

SINGLE STEP†	**SINGLE STEP** (Trap flag interrupt) Interrupt if TF = 1	Flags	O	D	I	T	S	Z	A	P	C
					0	0					

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	50	5	N/A	N/A

STC	**STC** (no operands) Set carry flag	Flags	O	D	I	T	S	Z	A	P	C
											1

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	2	—	1	STC

STD	**STD** (no operands) Set direction flag	Flags	O	D	I	T	S	Z	A	P	C
				1							

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	2	—	1	STD

STI	**STI** (no operands) Set interrupt enable flag	Flags	O	D	I	T	S	Z	A	P	C
					1						

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	2	—	1	STI

STOS	**STOS** dest-string Store byte or word string	Flags	O	D	I	T	S	Z	A	P	C

Operands	Clocks	Transfers*	Bytes	Coding Example
dest-string	11	1	1	STOS PRINT_LINE
(repeat) dest-string	9+10/rep	1/rep	1	REP STOS DISPLAY

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

†SINGLE STEP is not an instruction; it is included in table 2-21 only for timing information.

Table 2-21. Instruction Set Reference Data (Cont'd.)

SUB	**SUB** destination,source Subtraction	**Flags**	O	D	I	T	S	Z	A	P	C
			X				X	X	X	X	X

Operands	Clocks	Transfers*	Bytes	Coding Example
register, register	3	—	2	SUB CX, BX
register, memory	9+EA	1	2-4	SUB DX, MATH_TOTAL [SI]
memory, register	16+EA	2	2-4	SUB [BP+2], CL
accumulator, immediate	4	—	2-3	SUB AL, 10
register, immediate	4	—	3-4	SUB SI, 5280
memory, immediate	17+EA	2	3-6	SUB [BP].BALANCE, 1000

TEST	**TEST** destination,source Test or non-destructive logical and	**Flags**	O	D	I	T	S	Z	A	P	C
			0				X	X	U	X	0

Operands	Clocks	Transfers*	Bytes	Coding Example
register, register	3	—	2	TEST SI, DI
register, memory	9+EA	1	2-4	TEST SI, END_COUNT
accumulator, immediate	4	—	2-3	TEST AL, 00100000B
register, immediate	5	—	3-4	TEST BX, 0CC4H
memory, immediate	11+EA	—	3-6	TEST RETURN_CODE, 01H

WAIT	**WAIT** (no operands) Wait while $\overline{\text{TEST}}$ pin not asserted	**Flags**	O	D	I	T	S	Z	A	P	C

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	3 + 5n	—	1	WAIT

XCHG	**XCHG** destination,source Exchange	**Flags**	O	D	I	T	S	Z	A	P	C

Operands	Clocks	Transfers*	Bytes	Coding Example
accumulator, reg16	3	—	1	XCHG AX, BX
memory, register	17+EA	2	2-4	XCHG SEMAPHORE, AX
register, register	4	—	2	XCHG AL, BL

XLAT	**XLAT** source-table Translate	**Flags**	O	D	I	T	S	Z	A	P	C

Operands	Clocks	Transfers*	Bytes	Coding Example
source-table	11	1	1	XLAT ASCII_TAB

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Mnemonics © Intel, 1978

Table 2-21. Instruction Set Reference Data (Cont'd.)

XOR	**XOR** destination,source Logical exclusive or	**Flags**	O	D	I	T	S	Z	A	P	C
			0				X	X	U	X	0

Operands	Clocks	Transfers*	Bytes	Coding Example
register, register	3	—	2	XOR CX, BX
register, memory	9+EA	1	2-4	XOR CL, MASK_BYTE
memory, register	16+EA	2	2-4	XOR ALPHA [SI], DX
accumulator, immediate	4	—	2-3	XOR AL, 01000010B
register, immediate	4	—	3-4	XOR SI, 00C2H
memory, immediate	17+EA	2	3-6	XOR RETURN_CODE, 0D2H

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Mnemonics © Intel, 1978

Appendix C

Answers to Chapter Reviews

Answers Chapter 1 Review

1. translate, source
2. two, one, and zero
3. two's
4.
    - A. 10101001
    - B. 00001111
    - C. 01000001
    - D. 10101010
5.
    - A. 10101010
    - B. 00010000
    - C. 01000010
    - D. 10101011

6.

A. 9EH
B. E7H
C. FFH
D. 8C0FH

7.

A. 1111111110100001
B. 0010000001111000
C. 0111111111111100
D. 0000000000000011

8. So the assembler will not confuse the number with a label.
9. ASCII, BAUDOT
10.

A. 2
B. 16
C. 10

11. It contains the letter G, which is not a valid hexadecimal number.

Answers to Chapter 2 Review

1. False
2. False
3. False, MASM is the IBM Macro Assembler. It is not an editor.
4. False
5. Only when absolutely necessary is program code ever entered directly in machine language. Machine language entry of programs is tedious and error prone.
6. Symbols, absolute
7. Modular, understand, maintain, debug
8. space
9. comments
10. Comments help document the program flow, which makes the program easier to understand and maintain by others.

Answers to Chapter 3 Review

1. three
2. CPU, memory, I/O
3. CPU
4. two (EU and BIU)
5. Execution unit (EU), queue
6. 16
7. 16
8. 1,048,576 (one megabyte)
9. B
10. 65,536-byte segments
11. segment, offset
12. Temporary storage of variables and a place to store the instruction pointer and code segment register during procedure calls.
13. 65.536 (64K)
14. Maximum
15. In maximum mode, the 8088 uses a bus controller (8288) to control the address and data bus. In minimum mode, the 8088 is responsible for generating all the control signals required by the system. Maximum mode allows the 8088 to be used with co-processors such as the Intel 8087 math processor.
16. 8
17. 16
18. interrupt
19. A microprocessor requires memory and I/O. A microcomputer not only contains the CPU but also the memory and I/O sections of a computer.
20. 14
21. AX, BX, CX, DX
22. 2
23. 2
24. 4, CS, DS, ES, SS
25. False

26. CS and IP
27. False
28. Flag, CS, and IP
29. Disabled, will not

    Or you may have answered: Enabled, will
30. Direct, immediate, register indirect, base with offset
31. appendix, effective address

Answers to Chapter 4 Review

1. `ADD COUNTER,20`
2. six

3.
   ```
   CMP     COUNTER,20H
   JE      THERE
   ```
4. Clears the AX register (zeros it).
5. You cannot specify a constant, as the destination operand in an instruction.
6. Nothing. AX is stored at the memory location defined by the label FACEH. The use of labels such as this is confusing. If a zero preceded the label as in 0FACEH, the assembler would have generated an error, since 0FACEH represents a hexadecimal number and cannot be used as the destination operand of a MOV instruction.
7. POP

8.
   ```
   XOR     SI,SI       ;Zero pointer for first byte
   MOV     CX,64       ;Number of bytes to clear
   CLR: MOV COUNTER[SI],00H      ;Store zero
   INC SI      ;Increment the pointer
   LOOP CLR      ;Continue until CX = 0
   ```

9.
   ```
   LEA BX,COUNTER
   MOV BX,OFFSET COUNTER
   ```
10. LOOPE, LOOPZ

Answers to Chapter 5 Review

1. Segment
2. NEAR or FAR
3. A–G

   B–F

   C–K

   D–I

   E–J
4. = (equal)
5. PAGE

Answers to Chapter 6 Review

1. Sequence, instructions
2. Header (name), body, terminator
3. terminates
4. macro's name
5. unique
6.
```
MOV     AH,FUNCTION_NUMBER       ;MS-DOS function number
INT     21H                      ;MS-DOS function request.
```
7. A single name can be used to reference several statements required to define an MS-DOS function cell.

Answers to Chapter 8 Review

1. transfer area
2. buffer, disk
3. Sequential record

   Random record

   Random block

   (See the text for a description of each)
4. True
5. False
6. six
7. program

Answers to Chapter 9 Review

1. Basic Input/Output System
2. False
3. Four
4. 128,000
5. False. The graphics adapter allows for a 320 by 200 medium resolution graphics display.

Answers to Chapter 10 Review

1. UART
2. 8259
3. two-way
4. False
5. does not
6. 600 baud
7. 8250
8. False
9. 1 1 0 0 0 0 0 1
10. 04H
11. wait, DSR and CTS

Appendix D

Program Listings

Listing 2-1 CONFIGSY.ASM 355
Listing 2-2 CONFIGSY.LST 361
Listing 6-1 MACFLE.MAC 369
Listing 6-2 DOSEQU.EQU 396
Listing 6-3 KEYDSP 399
Listing 7-1 NUMBERSY.ASM 401
Listing 8-1 NEW_TYPE.ASM 412
Listing 8-2 NEW_COPY.ASM 416
Listing 8-3 FAST_CPY.ASM 421
Listing 8-4 DIRREAD.ASM 426
Listing 9-1 GRAPHIC.ASM 435
Listing 9-2 SOUND.ASM 440
Listing 10-1 DLOAD.ASM 443
Listing 10-2 COMM.ASM 447

Listing 2-1
CONFIGSY.ASM

```
        Page    56,132
        TITLE   ROUTINE TO RETURN CONFIGURATION OF SYSTEM
;** Written: 06-15-84
;** By: Gary A. Shade
;** (c) Gary A. Shade, 1984
;** Last Revised: 03-03-85
;
;** THIS ROUTINE WILL RETURN THE CURRENT CONFIGURATION
;** OF THE SYSTEM AND DISPLAY THE INFORMATION IN THE SCREEN
;
;** USE IT AS A STAND ALONE ROUTINE, OR AS A PROCEDURE
;** WHICH CAN BE USED TO RETURN STATUS TO A LARGER PROGRAM.
;USES BIOS CALLS INT 11H, AND INT 12H TO FETCH EQUIPMENT CONFIG.
;AND MEMORY SIZE.
;THIS PROCEDURE WILL NOT RECOGNIZE THE PRECENSE OF A
;WINCHESTER DISK IN THE SYSTEM. IT ALSO ASSUMES THAT THE
;SWITCHES SET ON THE PC MOTHERBOARD ARE CORRECTLY SET TO REFLECT
;THE NUMBER OF FLOPPY DISK DRIVES IN THE SYSTEM.
;
;SEE IBM TECHNICAL REFERENCE MANUAL SYSTEM BIOS, PAGE A-73
;FOR A DESCRIPTION OF THE BIT ASSIGNMENTS USED AS
;EQUIPMENT FLAGS (Also illustrated in Chapter 10).

MY_DATA SEGMENT PARA    'DATA'
MEM_SIZE        DW      ?       ;STORAGE FOR MEMORY SIZE
HDWR_TYPES      DW      ?       ;STORAGE FOR HARDWARE TYPES
TEMP1           DW      3 DUP(?) ;MISCELLANEOUS STORAGE
MEM_MESS        DB      'Memory Available                : ','$'
VIDEO_MODE      DB      'Initial Video Mode              : ','$'
TOTAL_DRIVES    DB      'Number of Disk Drives           : ','$'
RS232_PORTS     DB      'Number of RS-232 Cards          : ','$'
GAME_PORT       DB      'Game Port                       : ','$'
PRINTER         DB      'Number of Printers              : ','$'

MEM_64K         DB      '64K Bytes','$'
MEM_96K         DB      '96K Bytes','$'
MEM_128K        DB      '128K Bytes','$'
MEM_256K        DB      '256K Bytes','$'
MEM_320K        DB      '320K Bytes','$'
MEM_352K        DB      '352K Bytes','$'
MEM_384K        DB      '384K Bytes','$'
MEM_448K        DB      '448K Bytes','$'
MEM_512K        DB      '512K Bytes','$'
MEM_576K        DB      '576K Bytes','$'
MEM_608K        DB      '608K Bytes','$'
MEM_640K        DB      '640K Bytes','$'

VID1            DB      '40 X 25 BW Using Color Card','$'
```

Listing 2-1 continued

```
VID2            DB      '80 x 25 BW Using Color Card','$'
VID3            DB      '80 x 25 BW Using BW Card','$'
IN_SYS          DB      'In System','$'

MY_DATA ENDS                      ;END OF SEGMENT
;
;
MY_STACK        SEGMENT PARA    STACK   'STACK'
        DW      100 DUP(?)
TOP_OF_STACK    EQU     $         ;WHERE TO START STACK
MY_STACK        ENDS
;
;
;  DEFINE PROGRAM EQUATES
;
DISP_LINE       EQU     09H       ;MSDOS FUNCTION CODE
                                  ;TO DISPLAY LINE ->DX
DISP_CHAR       EQU     02H       ;MSDOS FUNCTION CODE
                                  ;TO DISPLAY CHAR IN DL

;
MY_CODE SEGMENT PARA    'CODE'
        ASSUME CS:MY_CODE,DS:MY_DATA,SS:MY_STACK,ES:MY_DATA
START   PROC    FAR               ;ENTRY POINT
INIT1:  PUSH    DS                ;SAVE OLD CODE SEGMENT
                                  ;PASSED IN DS
        XOR     AX,AX             ;SAVE A ZERO OFFSET
        PUSH    AX                ;FOR MSDOS RETURN
INIT2:  MOV     AX,MY_DATA        ;SET UP SEGMENT REGISTERS
        MOV     DS,AX
        MOV     ES,AX

;*********** Get the memory size in system. ****************

GET_MEM:        INT     12H       ;GET MEMORY SIZE IN K BYTES
        MOV     MEM_SIZE,AX       ;RETURNED IN AX REGISTER
GET_SYS:        INT     11H       ;GET SYSTEM CONFIGURATION
        MOV     HDWR_TYPES,AX     ;RETURNED IN AX
                                  ;A BIT SET MEANS
                                  ;THE ITEM IS PRESENT
                                  ;IN THE SYSTEM
;********** Clear the screen *******************************

CLR_SCREEN:     MOV     CX,24     ;SET UP LOOP COUNTER
CLS:    CALL    CRLF              ;DO CARRIAGE RETURN LINEFEED
        LOOP    CLS               ;FOR CX NUMBER OF TIMES

;********** Show available memory ***************************

MEM_SHOW:       LEA     DX,MEM_MESS     ;MEMORY MESSAGE
        MOV     AH,DISP_LINE      ;DISPLAY CODE FOR MSDOS
        INT     21H
        MOV     AX,MEM_SIZE       ;GET SIZE
        TEST    AH,00000010B      ;TEST B1 OF AH - IF SET
                                  ;MEM > 256k
        JNZ     M_BIG             ;
        TEST    AH,00000001B      ;TEST B8 OF AX (B0 OF AH)
        JNZ     M256K             ;256K SYSTEM
        TEST    AL,10000000B      ;TEST B7 OF AX IF SET
        JNZ     M128K               ;THEN 128 K SYSTEM
        TEST    AL,01100000B        ;TEST B6 AND B5, IF SET
        JNZ     M96K                ;THEN 96K SYSTEM
M64K:   LEA     DX,MEM_64K          ;DISPLAY MESSAGE FOR 64K
        CALL    DISP_MESS
        JMP     DCONFIG
M96K:   LEA     DX,MEM_96K          ;DISPLAY MESSAGE FOR 96K
        CALL    DISP_MESS
        JMP     DCONFIG
M128K:  LEA     DX,MEM_128K         ;DISPLAY 128K MESSAGE
        CALL    DISP_MESS
        JMP     DCONFIG

M256K:
        TEST    AL,11000000B        ;THEN 256+ 128 + 64 = 448K
        JNZ     M448K
        TEST    AL,10000000B        ;THEN 256k + 128k = 384k
        JNZ     M384k
        TEST    AL,01100000B        ;THEN 256K+96K = 352K
        JNZ     M352K
        TEST    AL,01000000B        ;THEN 256+64K = 320K
        JNZ     M320K

        LEA     DX,MEM_256K         ;DISPLAY MESSAGE FOR 256K
        CALL    DISP_MESS
        JMP     DCONFIG
M448K:
        LEA     DX,MEM_448K
        CALL    DISP_MESS
        JMP     DCONFIG

M320K:  LEA     DX,MEM_320K
        CALL    DISP_MESS
        JMP     DCONFIG

M352K:  LEA     DX,MEM_352K
        CALL    DISP_MESS           ;Display message
        JMP     DCONFIG

M384K:  LEA     DX,MEM_384K         ;Display the message for
        CALL    DISP_MESS
        JMP     DCONFIG

M_BIG:
        TEST    AL,10000000B        ;640k?
        JNZ     M640K
        TEST    AL,01100000B        ;608k?
        JNZ     M608K
        TEST    AL,01000000B        ;576k?
        JNZ     M576K
                                    ;Assume 512k
M512K:
        LEA     DX,MEM_512K         ;Display message for memory
                                    ;size.
        CALL    DISP_MESS
        JMP     DCONFIG
M576K:
        LEA     DX,MEM_576K

        CALL    DISP_MESS
        JMP     DCONFIG
```

Listing 2-1 continued

```
M608K:
        LEA     DX,MEM_608K
        CALL    DISP_MESS
        JMP     DCONFIG
M640K:
        LEA     DX,MEM_640K
        CALL    DISP_MESS

;****** Show the system configuration next. *********
;       Do video mode first.

DCONFIG:        LEA     DX,VIDEO_MODE   ;GET MESSAGE
        MOV     AH,DISP_LINE    ;DISPLAY THE LINE
        INT     21H
DC1:    MOV     AX,HDWR_TYPES   ;GET SYS CONFIG.
        MOV     CL,4            ;SHIFT BITS 4,5 TO 0,1
        ROR     AX,CL           ;TEST FOR VIDEO MODE
        AND     AL,00000011B    ;MASK BITS ONLY 0,1 IMPORTANT
        DEC     AL              ;IF 01, THEN 40 X 25 BLACK/WHITE
        JZ      BWC_4025        ;YES THEN GO DISPLAY IT
        DEC     AL              ;IF 02 THEN 80 X 25 B/W
        JZ      BWC_8025        ;YES THEN DISPLAY IT
BWCARD_8025:    LEA     DX,VID3 ;MUST BE 80 X 25 USING BW CARD
        CALL    DISP_MESS       ;Display the card type
        JMP     DC2             ;GO GET NEXT FUNCTION
BWC_8025:       LEA     DX,VID2 ;80 X 25 BW USING COLOR CARD
        CALL    DISP_MESS       ;Display the card type
        JMP     DC2             ;GET NEXT VALUE
BWC_4025:       LEA     DX,VID1 ;40 X 25 BW USING COLOR CARD
        CALL    DISP_MESS       ;DISPLAY THE CARD TYPE
;********** Now test for the number of disk drives int he system.

DC2:
        MOV     AX,HDWR_TYPES   ;GET STATUS BYTE
        ROR     AX,1            ;ROTATE B0 INTO CARRY
        MOV     TEMP1,AX        ;STOR VAL IN TEMP 1
        JNC     DC3             ;NOT IN SYSTEM, CONTINUE
        LEA     DX,TOTAL_DRIVES ;MESSAGE FOR DRIVES
        MOV     AH,DISP_LINE    ;MSDOS FUNCTION CODE
        INT     21H             ;MSDOS CALL
        MOV     AX,TEMP1        ;RECOVER VALUE
        MOV     CL,5            ;ROTATE 5 MORE BITS
                                ;GET BITS 6,7 TO 0,1
        ROR     AX,CL           ;ROTATE 5 TIMES
        AND     AL,00000011B    ;MASK BITS
        INC     AL              ;ADD 1 TO VALUE
        OR      AL,30H          ;CONVERT TO ASCII
        MOV     DL,AL           ;SET UP VALUE TO DISPLAY
        MOV     AH,DISP_CHAR    ;MSDOS FUNCTION CODE
        INT     21H             ;MSDOS CALL

;********** How many RS-232C serial ports in system?

DC3:    CALL    CRLF            ;NEWLINE
        LEA     DX,RS232_PORTS  ;OPTION IN SYSTEM?
        MOV     AH,DISP_LINE    ;DISPLAY FUNCTION CODE
        INT     21H             ;MSDOS CALL
        MOV     AX,HDWR_TYPES
        MOV     CL,9            ;ROTATE 9 TIMES
        ROR     AX,CL           ;BITS 9,10,11 TO 0,1,2
        AND     AL,00000111B    ;MASK BITS
        OR      AL,30H          ;CONVERT TO ASCII
        MOV     DL,AL           ;SET UP DL WITH CHAR TO DISP
        MOV     AH,DISP_CHAR    ;MSDOS FUNCTION CODE
        INT     21H             ;MSDOS CALL

;*********** Is the game port in the system?

DC4:    CALL    CRLF            ;NEWLINE
        LEA     DX,GAME_PORT    ;GAME PORT IN SYS?
        MOV     AH,DISP_LINE    ;FUNCTION CODE FOR DSPL
        INT     21H
        MOV     AX,HDWR_TYPES   ;GET CONFIGURATION CODE
        MOV     CL,13           ;INITIALIZE ROTATE COUNT
        RCR     AX,CL           ;IN SYSTEM?
        JNC     DC5             ;IF NOT, CONTINUE.
        CALL    YES_IN_SYS      ;SAY IT IS ON DISPLAY

;******* How about printers? How many?

DC5:    CALL    CRLF            ;NEWLINE
        LEA     DX,PRINTER      ;HOW MANY PRINTERS?
        MOV     AH,DISP_LINE    ;FUNCTION CODE FOR DSP
        INT     21H
        MOV     AX,HDWR_TYPES   ;GET STATUS WORD
        MOV     CL,14           ;INITIALIZE ROTATE COUNT
        ROR     AX,CL           ;IS IT IN THE SYS?
        AND     AL,00000011B    ;MASK BITS
        OR      AL,30H          ;CONVERT TO ASCII
        MOV     DL,AL           ;SET UP DL FOR DISPLAY
        MOV     AH,DISP_CHAR    ;MSDOS FUNCTION CALL
        INT     21H
DONE:   RET
START   ENDP
;************************************************************
;Procedure to show that the resource is available.
;************************************************************

YES_IN_SYS      PROC    NEAR
        LEA     DX,IN_SYS       ;SHOW THAT IS IN THE SYSTEM
        MOV     AH,DISP_LINE    ;MSDOS FUNCTION CODE
                                ;FOR DISPLAYING LINE
                                ;POINTED TO BY DX
        INT     21H             ;MSDOS FUNCTION CALL
        RET                     ;RETURN TO CALLER
YES_IN_SYS      ENDP
;****************************************************
;Carriage return linefeed procedure. Use a similar procedure,
;or this one to generate a carriage return and a line feed
;to move the cursor to the 1st column of the next line
;(also known as a new line function).
;*****************************************************

CRLF    PROC    NEAR            ;NEWLINE FUNCTION FOR
                                ;VIDEO
        MOV     DL,0AH          ;LINEFEED CODE
        MOV     AH,DISP_CHAR    ;MSDOS FUNCTION CODE
                                ;TO DISPLAY CHARACTER
        INT     21H             ;MSDOS CALL
```

Listing 2-1 continued

```
        MOV     DL,0DH          ;CARRIAGE RETURN CODE
        MOV     AH,DISP_CHAR    ;DISPLAY THE CHARACTER
        INT     21H
        RET
CRLF    ENDP


;**** PROCEDURE TO DISPLAY MEMORY SIZE - DX MUST BE POINTING
;**** TO THE BEGINNING OF A MESSAGE STRING ON ENTRY.

DISP_MESS       PROC    NEAR
        MOV     AH,DISP_LINE    ;384K of RAM in system
        INT     21H
        CALL    CRLF
        RET
DISP_MESS       ENDP
;*****************************************************

MY_CODE ENDS
        END     INIT1
```

Listing 2-2
CONFIGSY.LST

```
 The IBM Personal Computer MACRO Assembler 03-03-85
ROUTINE TO RETURN CONFIGURATION OF SYSTEM


                                        Page    56,132
                                        TITLE   ROUTINE TO RETURN CONFIGURATION OF SYSTEM
                                ;** Written: 06-15-84
                                ;** By: Gary A. Shade
                                ;** (c) Gary A. Shade, 1984
                                ;** Last Revised: 03-03-85
                                ;
                                ;** THIS ROUTINE WILL RETURN THE CURRENT CONFIGURATION
                                ;** OF THE SYSTEM AND DISPLAY THE INFORMATION IN THE SCREEN
                                ;
                                ;** USE IT AS A STAND ALONE ROUTINE, OR AS A PROCEDURE
                                ;** WHICH CAN BE USED TO RETURN STATUS TO A LARGER PROGRAM.
                                ;USES BIOS CALLS INT 11H, AND INT 12H TO FETCH EQUIPMENT CONFIG.
                                ;AND MEMORY SIZE.
                                ;THIS PROCEDURE WILL NOT RECOGNIZE THE PRECENSE OF A
                                ;WINCHESTER DISK IN THE SYSTEM. IT ALSO ASSUMES THAT THE
                                ;SWITCHES SET ON THE PC MOTHERBOARD ARE CORRECTLY SET TO REFLECT
                                ;THE NUMBER OF FLOPPY DISK DRIVES IN THE SYSTEM.
                                ;
                                ;SEE IBM TECHNICAL REFERENCE MANUAL SYSTEM BIOS, PAGE A-73
                                ;FOR A DESCRIPTION OF THE BIT ASSIGNMENTS USED AS
                                ;EQUIPMENT FLAGS (Also illustrated in Chapter 10).

0000                            MY_DATA SEGMENT PARA    'DATA'
0000  ????                      MEM_SIZE        DW      ?       ;STORAGE FOR MEMORY SIZE
0002  ????                      HDWR_TYPES      DW      ?       ;STORAGE FOR HARDWARE TYPES
0004     03 [                   TEMP1           DW      3 DUP(?) ;MISCELLANEOUS STORAGE
                ????
                      ]

000A  4D 65 6D 6F 72 79         MEM_MESS        DB      'Memory Available                 : ','$'
      20 41 76 61 69 6C
      61 62 6C 65 20 20
      20 20 20 20 20 20
      20 20 20 20 20 20
      20 20 3A 20 24
002D  49 6E 69 74 69 61         VIDEO_MODE      DB      'Initial Video Mode               : ','$'
```

Listing 2-2 continued

```
      6C 20 56 69 64 65
      6F 20 4D 6F 64 65
      20 20 20 20 20 20
      20 20 20 20 20 20
      20 20 3A 20 24
0050  4E 75 6D 62 65 72      TOTAL_DRIVES  DB     'Number of Disk Drives              : ','$'
      20 6F 66 20 44 69
      73 6B 20 44 72 69
      76 65 73 20 20 20
      20 20 20 20 20 20
      20 20 3A 20 24
0073  4E 75 6D 62 65 72      RS232_PORTS   DB     'Number of RS-232 Cards             : ','$'
      20 6F 66 20 52 53
      2D 32 33 32 20 43
      61 72 64 73 20 20
      20 20 20 20 20 20
      20 20 3A 20 24
0096  47 61 6D 65 20 50      GAME_PORT     DB     'Game Port                          : ','$'
      6F 72 74 20 20 20
      20 20 20 20 20 20
      20 20 20 20 20 20
      20 20 20 20 20 20
      20 20 3A 20 24
00B9  4E 75 6D 62 65 72      PRINTER       DB     'Number of Printers                 : ','$'
      20 6F 66 20 50 72
      69 6E 74 65 72 73
      20 20 20 20 20 20
      20 20 20 20 20 20
      20 20 3A 20 24

00DC  36 34 4B 20 42 79      MEM_64K       DB     '64K Bytes','$'
      74 65 73 24
00E6  39 36 4B 20 42 79      MEM_96K       DB     '96K Bytes','$'
      74 65 73 24
00F0  31 32 38 4B 20 42      MEM_128K      DB     '128K Bytes','$'
      79 74 65 73 24
00FB  32 35 36 4B 20 42      MEM_256K      DB     '256K Bytes','$'
      79 74 65 73 24
0106  33 32 30 4B 20 42      MEM_320K      DB     '320K Bytes','$'
      79 74 65 73 24
0111  33 35 32 4B 20 42      MEM_352K      DB     '352K Bytes','$'
      79 74 65 73 24
011C  33 38 34 4B 20 42      MEM_384K      DB     '384K Bytes','$'
      79 74 65 73 24
0127  34 34 38 4B 20 42      MEM_448K      DB     '448K Bytes','$'
      79 74 65 73 24
0132  35 31 32 4B 20 42      MEM_512K      DB     '512K Bytes','$'
      79 74 65 73 24
013D  35 37 36 4B 20 42      MEM_576K      DB     '576K Bytes','$'
      79 74 65 73 24
```

```
0148  36 30 38 4B 20 42          MEM_608K      DB      '608K Bytes','$'
      79 74 65 73 24
0153  36 34 30 4B 20 42          MEM_640K      DB      '640K Bytes','$'
      79 74 65 73 24

015E  34 30 20 58 20 32          VID1          DB      '40 X 25 BW Using Color Card','$'
      35 20 42 57 20 55
      73 69 6E 67 20 43
      6F 6C 6F 72 20 43
      61 72 64 24
017A  38 30 20 78 20 32          VID2          DB      '80 x 25 BW Using Color Card','$'
      35 20 42 57 20 55
      73 69 6E 67 20 43
      6F 6C 6F 72 20 43
      61 72 64 24
0196  38 30 20 78 20 32          VID3          DB      '80 x 25 BW Using BW Card','$'
      35 20 42 57 20 55
      73 69 6E 67 20 42
      57 20 43 61 72 64
      24
01AF  49 6E 20 53 79 73          IN_SYS        DB      'In System','$'
      74 65 6D 24

01B9                             MY_DATA ENDS                    ;END OF SEGMENT
                                 ;
                                 ;
0000                             MY_STACK      SEGMENT PARA    STACK   'STACK'
0000     64 [                            DW    100 DUP(?)
                 ????
                         ]

= 00C8                           TOP_OF_STACK  EQU     $         ;WHERE TO START STACK
00C8                             MY_STACK      ENDS
                                 ;
                                 ;
                                 ;  DEFINE PROGRAM EQUATES
                                 ;
= 0009                           DISP_LINE     EQU     09H       ;MSDOS FUNCTION CODE
                                                                 ;TO DISPLAY LINE ->DX
= 0002                           DISP_CHAR     EQU     02H       ;MSDOS FUNCTION CODE
                                                                 ;TO DISPLAY CHAR IN DL


                                 ;
0000                             MY_CODE SEGMENT PARA    'CODE'
                                         ASSUME CS:MY_CODE,DS:MY_DATA,SS:MY_STACK,ES:MY_DATA
0000                             START   PROC    FAR             ;ENTRY POINT
0000  1E                         INIT1:  PUSH    DS              ;SAVE OLD CODE SEGMENT
                                                                 ;PASSED IN DS
0001  33 C0                              XOR     AX,AX           ;SAVE A ZERO OFFSET
0003  50                                 PUSH    AX              ;FOR MSDOS RETURN
0004  B8 ---- R                  INIT2:  MOV     AX,MY_DATA      ;SET UP SEGMENT REGISTERS
0007  8E D8                              MOV     DS,AX
```

Listing 2-2 continued

```
0009  8E C0                    MOV     ES,AX

                      ;*********** Get the memory size in system. ****************

000B  CD 12           GET_MEM:          INT      12H      ;GET MEMORY SIZE IN K BYTES
000D  A3 0000 R                MOV     MEM_SIZE,AX        ;RETURNED IN AX REGISTER
0010  CD 11           GET_SYS:          INT      11H      ;GET SYSTEM CONFIGURATION
0012  A3 0002 R                MOV     HDWR_TYPES,AX      ;RETURNED IN AX
                                                          ;A BIT SET MEANS
                                                          ;THE ITEM IS PRESENT
                                                          ;IN THE SYSTEM
                      ;********** Clear the screen *******************************

0015  B9 0018         CLR_SCREEN:       MOV      CX,24    ;SET UP LOOP COUNTER
0018  E8 017D R       CLS:     CALL    CRLF               ;DO CARRIAGE RETURN LINEFEED
001B  E2 FB                    LOOP    CLS                ;FOR CX NUMBER OF TIMES

                      ;********** Show available memory ****************************

001D  8D 16 000A R    MEM_SHOW:         LEA      DX,MEM_MESS      ;MEMORY MESSAGE
0021  B4 09                    MOV     AH,DISP_LINE       ;DISPLAY CODE FOR MSDOS
0023  CD 21                    INT     21H
0025  A1 0000 R                MOV     AX,MEM_SIZE        ;GET SIZE
0028  F6 C4 02                 TEST    AH,00000010B       ;TEST B1 OF AH - IF SET
                                                          ;MEM > 256k
002B  75 6D                    JNZ     M_BIG              ;
002D  F6 C4 01                 TEST    AH,00000001B       ;TEST B8 OF AX (B0 OF AH)
0030  75 26                    JNZ     M256K              ;256K SYSTEM
0032  A8 80                    TEST    AL,10000000B       ;TEST B7 OF AX IF SET
0034  75 18                    JNZ     M128K              ;THEN 128 K SYSTEM
0036  A8 60                    TEST    AL,01100000B       ;TEST B6 AND B5, IF SET
0038  75 0A                    JNZ     M96K               ;THEN 96K SYSTEM
003A  8D 16 00DC R    M64K:    LEA     DX,MEM_64K         ;DISPLAY MESSAGE FOR 64K
003E  E8 018A R                CALL    DISP_MESS
0041  E9 00CB R                JMP     DCONFIG
0044  8D 16 00E6 R    M96K:    LEA     DX,MEM_96K         ;DISPLAY MESSAGE FOR 96K
0048  E8 018A R                CALL    DISP_MESS
004B  EB 7E 90                 JMP     DCONFIG
004E  8D 16 00F0 R    M128K:   LEA     DX,MEM_128K        ;DISPLAY 128K MESSAGE
0052  E8 018A R                CALL    DISP_MESS
0055  EB 74 90                 JMP     DCONFIG

0058                  M256K:
0058  A8 C0                    TEST    AL,11000000B       ;THEN 256+ 128 + 64 = 448K
005A  75 16                    JNZ     M448K
005C  A8 80                    TEST    AL,10000000B       ;THEN 256k + 128k = 384k
005E  75 30                    JNZ     M384k
0060  A8 60                    TEST    AL,01100000B       ;THEN 256K+96K = 352K
0062  75 22                    JNZ     M352K
0064  A8 40                    TEST    AL,01000000B       ;THEN 256+64K = 320K
0066  75 14                    JNZ     M320K
```

```

 0068  8D 16 00FB R                 LEA     DX,MEM_256K      ;DISPLAY MESSAGE FOR 256K
 006C  E8 018A R                    CALL    DISP_MESS
 006F  EB 5A 90                     JMP     DCONFIG
 0072                       M448K:
 0072  8D 16 0127 R                 LEA     DX,MEM_448K
 0076  E8 018A R                    CALL    DISP_MESS
 0079  EB 50 90                     JMP     DCONFIG

 007C  8D 16 0106 R         M320K:  LEA     DX,MEM_320K
 0080  E8 018A R                    CALL    DISP_MESS
 0083  EB 46 90                     JMP     DCONFIG

 0086  8D 16 0111 R         M352K:  LEA     DX,MEM_352K
 008A  E8 018A R                    CALL    DISP_MESS        ;Display message
 008D  EB 3C 90                     JMP     DCONFIG

 0090  8D 16 011C R         M384K:  LEA     DX,MEM_384K      ;Display the message for
 0094  E8 018A R                    CALL    DISP_MESS
 0097  EB 32 90                     JMP     DCONFIG

 009A                       M_BIG:
 009A  A8 80                        TEST    AL,10000000B     ;640k?
 009C  75 26                        JNZ     M640K
 009E  A8 60                        TEST    AL,01100000B     ;608k?
 00A0  75 18                        JNZ     M608K
 00A2  A8 40                        TEST    AL,01000000B     ;576k?
 00A4  75 0A                        JNZ     M576K
                                                             ;Assume 512k
 00A6                       M512K:
 00A6  8D 16 0132 R                 LEA     DX,MEM_512K      ;Display message for memory
                                                             ;size.
 00AA  E8 018A R                    CALL    DISP_MESS
 00AD  EB 1C 90                     JMP     DCONFIG
 00B0                       M576K:
 00B0  8D 16 013D R                 LEA     DX,MEM_576K
 00B4  E8 018A R                    CALL    DISP_MESS
 00B7  EB 12 90                     JMP     DCONFIG
 00BA                       M608K:
 00BA  8D 16 0148 R                 LEA     DX,MEM_608K
 00BE  E8 018A R                    CALL    DISP_MESS
 00C1  EB 08 90                     JMP     DCONFIG
 00C4                       M640K:
 00C4  8D 16 0153 R                 LEA     DX,MEM_640K
 00C8  E8 018A R                    CALL    DISP_MESS

                            ;****** Show the system configuration next. *********
                            ;       Do video mode first.

 00CB  8D 16 002D R         DCONFIG:        LEA     DX,VIDEO_MODE   ;GET MESSAGE
 00CF  B4 09                        MOV     AH,DISP_LINE     ;DISPLAY THE LINE
 00D1  CD 21                        INT     21H
```

Listing 2-2 continued

```
00D3  A1 0002 R         DC1:        MOV     AX,HDWR_TYPES   ;GET SYS CONFIG.
00D6  B1 04                         MOV     CL,4            ;SHIFT BITS 4,5 TO 0,1
00D8  D3 C8                         ROR     AX,CL           ;TEST FOR VIDEO MODE
00DA  24 03                         AND     AL,00000011B    ;MASK BITS ONLY 0,1 IMPORTANT
00DC  FE C8                         DEC     AL              ;IF 01, THEN 40 X 25 BLACK/WHITE
00DE  74 18                         JZ      BWC_4025        ;YES THEN GO DISPLAY IT
00E0  FE C8                         DEC     AL              ;IF 02 THEN 80 X 25 B/W
00E2  74 0A                         JZ      BWC_8025        ;YES THEN DISPLAY IT
00E4  8D 16 0196 R      BWCARD_8025:        LEA     DX,VID3 ;MUST BE 80 X 25 USING BW CARD
00E8  E8 018A R                     CALL    DISP_MESS       ;Display the card type
00EB  EB 12 90                      JMP     DC2             ;GO GET NEXT FUNCTION
00EE  8D 16 017A R      BWC_8025:           LEA     DX,VID2 ;80 X 25 BW USING COLOR CARD
00F2  E8 018A R                     CALL    DISP_MESS       ;Display the card type
00F5  EB 08 90                      JMP     DC2             ;GET NEXT VALUE
00F8  8D 16 015E R      BWC_4025:           LEA     DX,VID1 ;40 X 25 BW USING COLOR CARD
00FC  E8 018A R                     CALL    DISP_MESS       ;DISPLAY THE CARD TYPE
                        ;********** Now test for the number of disk drives int he system.

00FF                    DC2:
00FF  A1 0002 R                     MOV     AX,HDWR_TYPES   ;GET STATUS BYTE
0102  D1 C8                         ROR     AX,1            ;ROTATE B0 INTO CARRY
0104  A3 0004 R                     MOV     TEMP1,AX        ;STOR VAL IN TEMP 1
0107  73 1B                         JNC     DC3             ;NOT IN SYSTEM, CONTINUE
0109  8D 16 0050 R                  LEA     DX,TOTAL_DRIVES ;MESSAGE FOR DRIVES
010D  B4 09                         MOV     AH,DISP_LINE    ;MSDOS FUNCTION CODE
010F  CD 21                         INT     21H             ;MSDOS CALL
0111  A1 0004 R                     MOV     AX,TEMP1        ;RECOVER VALUE
0114  B1 05                         MOV     CL,5            ;ROTATE 5 MORE BITS
                                                            ;GET BITS 6,7 TO 0,1
0116  D3 C8                         ROR     AX,CL           ;ROTATE 5 TIMES
0118  24 03                         AND     AL,00000011B    ;MASK BITS
011A  FE C0                         INC     AL              ;ADD 1 TO VALUE
011C  0C 30                         OR      AL,30H          ;CONVERT TO ASCII
011E  8A D0                         MOV     DL,AL           ;SET UP VALUE TO DISPLAY
0120  B4 02                         MOV     AH,DISP_CHAR    ;MSDOS FUNCTION CODE
0122  CD 21                         INT     21H             ;MSDOS CALL

                        ;********** How many RS-232C serial ports in system?

0124  E8 017D R         DC3:        CALL    CRLF            ;NEWLINE
0127  8D 16 0073 R                  LEA     DX,RS232_PORTS  ;OPTION IN SYSTEM?
012B  B4 09                         MOV     AH,DISP_LINE    ;DISPLAY FUNCTION CODE
012D  CD 21                         INT     21H             ;MSDOS CALL
012F  A1 0002 R                     MOV     AX,HDWR_TYPES
0132  B1 09                         MOV     CL,9            ;ROTATE 9 TIMES
0134  D3 C8                         ROR     AX,CL           ;BITS 9,10,11 TO 0,1,2
0136  24 07                         AND     AL,00000111B    ;MASK BITS
0138  0C 30                         OR      AL,30H          ;CONVERT TO ASCII
013A  8A D0                         MOV     DL,AL           ;SET UP DL WITH CHAR TO DISP
013C  B4 02                         MOV     AH,DISP_CHAR    ;MSDOS FUNCTION CODE
013E  CD 21                         INT     21H             ;MSDOS CALL
```

```
                               ;*********** Is the game port in the system?

0140  E8 017D R                DC4:   CALL   CRLF           ;NEWLINE
0143  8D 16 0096 R                    LEA    DX,GAME_PORT   ;GAME PORT IN SYS?
0147  B4 09                           MOV    AH,DISP_LINE   ;FUNCTION CODE FOR DSPL
0149  CD 21                           INT    21H
014B  A1 0002 R                       MOV    AX,HDWR_TYPES  ;GET CONFIGURATION CODE
014E  B1 0D                           MOV    CL,13          ;INITIALIZE ROTATE COUNT
0150  D3 D8                           RCR    AX,CL          ;IN SYSTEM?
0152  73 03                           JNC    DC5            ;IF NOT, CONTINUE.
0154  E8 0174 R                       CALL   YES_IN_SYS     ;SAY IT IS ON DISPLAY

                               ;******* How about printers? How many?

0157  E8 017D R                DC5:   CALL   CRLF           ;NEWLINE
015A  8D 16 00B9 R                    LEA    DX,PRINTER     ;HOW MANY PRINTERS?
015E  B4 09                           MOV    AH,DISP_LINE   ;FUNCTION CODE FOR DSP
0160  CD 21                           INT    21H
0162  A1 0002 R                       MOV    AX,HDWR_TYPES  ;GET STATUS WORD
0165  B1 0E                           MOV    CL,14          ;INITIALIZE ROTATE COUNT
0167  D3 C8                           ROR    AX,CL          ;IS IT IN THE SYS?
0169  24 03                           AND    AL,00000011B   ;MASK BITS
016B  0C 30                           OR     AL,30H         ;CONVERT TO ASCII
016D  8A D0                           MOV    DL,AL          ;SET UP DL FOR DISPLAY
016F  B4 02                           MOV    AH,DISP_CHAR   ;MSDOS FUNCTION CALL
0171  CD 21                           INT    21H
0173  CB                       DONE:  RET
0174                           START  ENDP
                               ;***********************************************************
                               ;Procedure to show that the resource is available.
                               ;***********************************************************

0174                           YES_IN_SYS    PROC   NEAR
0174  8D 16 01AF R                    LEA    DX,IN_SYS      ;SHOW THAT IS IN THE SYSTEM
0178  B4 09                           MOV    AH,DISP_LINE   ;MSDOS FUNCTION CODE
                                                            ;FOR DISPLAYING LINE
                                                            ;POINTED TO BY DX
017A  CD 21                           INT    21H            ;MSDOS FUNCTION CALL
017C  C3                              RET                   ;RETURN TO CALLER
017D                           YES_IN_SYS    ENDP
                               ;***************************************************
                               ;Carriage return linefeed procedure. Use a similar procedure,
                               ;or this one to generate a carriage return and a line feed
                               ;to move the cursor to the 1st column of the next line
                               ;(also known as a new line function).
                               ;****************************************************

017D                           CRLF   PROC   NEAR           ;NEWLINE FUNCTION FOR
                                                            ;VIDEO
017D  B2 0A                           MOV    DL,0AH         ;LINEFEED CODE
017F  B4 02                           MOV    AH,DISP_CHAR   ;MSDOS FUNCTION CODE
```

Listing 2-2 continued

```
                                                   ;TO DISPLAY CHARACTER
0181  CD 21                      INT     21H       ;MSDOS CALL
0183  B2 0D                      MOV     DL,0DH    ;CARRIAGE RETURN CODE
0185  B4 02                      MOV     AH,DISP_CHAR  ;DISPLAY THE CHARACTER
0187  CD 21                      INT     21H
0189  C3                         RET
018A                     CRLF    ENDP


                         ;**** PROCEDURE TO DISPLAY MEMORY SIZE - DX MUST BE POINTING
                         ;**** TO THE BEGINNING OF A MESSAGE STRING ON ENTRY.

018A                     DISP_MESS       PROC    NEAR
018A  B4 09                      MOV     AH,DISP_LINE    ;384K of RAM in system
018C  CD 21                      INT     21H
018E  E8 017D R                  CALL    CRLF
0191  C3                         RET
0192                     DISP_MESS       ENDP
                         ;*******************************************************

0192                     MY_CODE ENDS
                                 END     INIT1
```

Listing 6-1
MACFLE.MAC

```
;***** This file will establish functional calls which can be
;invoked through the use of a macro call. The Macros will
;utilize the necessary registers, without saving the register's
;previous contents. Therefore it is necessary for the programmer
;to save any usable data prior to invoking the macro.
;See  the  DOS Technical Reference  Manual  (MSDOS  or  PCDOS)  for
;register usage. You must also INCLUDE DOSEQU.EQU when using this file.
;
;Created 03/17/84
;Last Updated: 10/03/84
;By: Gary A. Shade


BIOS    EQU     10H     ;BIOS VIDEO INTERRUPT.
MSDOS   EQU     21H     ;MSDOS INTERRUPT

;********************************************************
;THIS MACRO WILL TERMINATE A USER PROGRAM.
;CS REGISTER MUST BE SET TO THE SEGMENT ADDRESS IN THE PSP
;PRIOR TO INVOKING THIS CALL. SEE IBM TECHNICAL REFERENCE
;MANUAL PAGE 5-17.
;Function 0
;
@TERMINATE   MACRO
     MOV  AH,F_TERMINATE              ;TERMINATE USER PROGRAM
     INT  MSDOS                       ;MSDOS CALL
     ENDM                             ;
;********************************************************
;THIS MACRO WILL WAIT FOR A KEY TO BE PRESSED AND ECHOES THE
;CHARACTER TO THE DISPLAY. ON RETURN THE CHARACTER IS IN AL
;IF A CONTROL-C IS TYPED, AN INT 23H IS EXECUTED.
;Function 1
;
@WAITKEY   MACRO
     MOV  AH,F_WAITKEY                ;FUNCTION REQUEST
     INT  MSDOS                       ;DOS CALL
     ENDM
;********************************************************
;THIS MACRO WILL DISPLAY THE CHARACTER IN DL.
;
;Function 2
;
@CHARDSP  MACRO
     MOV  AH,F_CHARDSP                ;DOS FUNCTION CODE
     INT  MSDOS                       ;
     ENDM
;********************************************************
;THIS MACRO WILL WAIT FOR A CHARACTER TO BE INPUT FROM THE
```

Listing 6-1 continued

```
;AUX DEVICE.
;Function 3
;
@WAITAUX  MACRO
     MOV  AH,F_WAITAUX           ;FUNCTION CODE
     INT  MSDOS                  ;DOS CALL
     ENDM
;********************************************************
;This macro will send the characterr in DL to the AUX device.
;
;Function 4
;
@AUXOUT MACRO
     MOV  AH,F_AUXOUT
     INT  MSDOS                  ;DOS CALL
     ENDM
;********************************************************
;This macro will output a character in DL to the printer
;if one is attached to the system.
;The character must be in DL before the macro is invoked.
;
;Function 5
;
@PRINTER_OUT    MACRO
        MOV     AH,F_PRINTER_OUT ;MSDOS FUNCTION CODE
        INT     MSDOS            ;SYSTEM CALL
        ENDM
;********************************************************
;This macro will return a keyboard character if one is ready.
;If DL = 0FFH then the routine will return a character form the
;Keyboard, if one is available (If AL= 00H on return,
;no character was available. If DL does not = 0FFH,
;then the routine will display the character in DL.
;Function 6
;
@CON_IO MACRO
        MOV     AH,F_CON_IO      ;MSDOS FUNCTION CODE
        INT     MSDOS            ;SYSTEM CALL
        ENDM
;*******************************************************
;This macro will wait for a character from the keyboard.
;The routine does not check for a control - C.
;The character is returned in AL and is not echoed to the display.
;
;Function 7
;
@CON_INPUT      MACRO
        MOV     AH,F_CON_INPUT
        INT     MSDOS
        ENDM
;*******************************************************
;This macro will wait for a keyboard character, and return
;it in AL without echoing the character to the display.
;If a control -c is pressed, the routine executes
;an interrupt (INT), 23H.
;
;Function 8
;
@CON_INPUT2     MACRO
```

```
         MOV     AH,F_CON_INPUT2 ;MSDOS FUNCTION CODE
         INT     MSDOS           ;MSDOS FUNCTION CALL
         ENDM
;*********************************************************
;Macro to display a line of information on the video display.
;Uses DX and AH registers.
;The  label  of  the message must be passed to  the  macro  as  a
;parameter.
;The string must be terminated with a '$' (24H)
;All function codes (F_xxx) can be found in the equate file
;dos.equ.
;
;
;Function 9
;
@VDLINE   MACRO     MESSAGE
          MOV  AH,F_VDLINE         ;FUNCTION CODE FOR
                                   ;LINE DISPLAY
          LEA  DX,MESSAGE
          INT  MSDOS               ;MSDOS CALL
          ENDM
;*********************************************************
;MACRO TO ACCEPT A LINE OF INPUT FROM THE KEYBOARD TERMINATED
;BY A CARRIAGE RETURN.
;A KEYBOARD BUFFER MUST BE ESTABLISHED WHICH IS DEFINED AS
;FOLLOWS: BYTE 0 = LENGTH OF THE BUFFER.
;         BYTE 2 = CHARACTERS ENTERED (RETUNED VALUE )
;         BYTE 3 THROUGH N = BUFFER, WHERE N IS LAST BYTE
;
;Function 0AH
;
@KBDLINE  MACRO     KBDBUFFER
          MOV  AH,F_KBDLINE        ;GET DOS FUNCTION CALL
          LEA  DX,KBDBUFFER        ;GET ADDRESS OF KBUFFER
          INT  MSDOS               ;EXECUTE CALL
          ENDM
;*********************************************************
;This macro invokes a routine to check for an available
;character from the keyboard's type ahead buffer. On return
;AL = 0FFH, means there are characters in the buffer.
;AL = 00H then there are no characters in the buffer.
;If the routine encounters a control-C, then INT 23H is
;executed.
;
;Function 0BH
;
@PEEK_BUFFER    MACRO
        MOV     AH,F_PEEK_BUFFER           ;MSDOS FUNCTION CODE
        INT     MSDOS                      ;SYS. CALL
        ENDM
;*********************************************************
;This macro will clear the type ahead buffer and invoke
;the function whose code is in the AL register at entry.
;The allowable functions are: 1, 6, 7, 8, 0A. If AL
;contains any other value, only the buffer is cleared, and
;the routine takes no other action.
;On return AL will contain 00H,if the buffer was cleared,
;and there was no other function executed.
;
;Function 0CH
```

Listing 6-1 continued

```
;
@KBD_FLUSH      MACRO FUNCTION
        MOV     AH,F_KBD_FLUSH              ;FUNCTION CODE
        MOV     AL,FUNCTION                 ;READ FUNCTION
                                            ;IF ANY.
        INT     MSDOS                       ;DO THE CALL
        ENDM
;*********************************************************
;This macro will invoke the routine which flushes all file
;buffers by writing modified files to disk. The directory
;is not updated, use the close function to update
;the directory properly.
;Function 0DH
;
@DISK_RESET     MACRO
        MOV     AH,F_DISK_RESET ;FUNCTION CODE
        INT     MSDOS           ;SYS. CALL
        ENDM
;*********************************************************
;This macro will select the specified disk drive as the
;default disk drive. The logical drive number is in DL
;[0= drive A, 1= drive B etc.]. On return AL contains the
;total number of disk drives in the system. See also BIOS
;interrupt 11H which returns the equipment configuration for
;your computer.
;The DRIVE_NUMBER parameter must be an capital ASCII letter
;'A' or 'B' etc. do not specify the drive number in lower
;case.
;Function 0EH
;
@SELECT_DISK    MACRO   DRIVE_NUMBER
        MOV     AH,F_SELECT_DISK            ;FUNCTION CODE
        MOV     DL,DRIVE_NUMBER[-40H]       ;DRIVE LETTER
        INT     MSDOS                       ;SYS. CALL
        ENDM
;*********************************************************
;This macro will invoke the MSDOS function which will open
;a disk file control block. DS:DX must point to the FCB.
;the FCB will be initialized as follows (assuming the file
;is found in the directory);
;BYTE in FCB            Initialized to:
;   0C-0D                  0000H   (Current Block Number)
;   0E-0F                  0080H   (Record Size)
;   10H-16H             Set by directory information.
;(File Size (10H), date last updated (14H), and time last
;updated (16H), are set from information contained
;in the directory.
;After opening the file, set bytes 0EH-0FH in the FCB to
;the desired record size,
;byte 20H to the current record number (sequential access),
;byte 21H must be set to the relative record
;number (random access).
;On exit; AL = 00H then directory entry was found.
;         AL = 0FFH then directory entry not found.
;
;Function 0FH
;
@OPEN   MACRO   FCB
        LEA     DX,FCB          ;Get the FCB.
        MOV     AH,F_OPEN        ;MSDOS FUNCTION CODE.
        INT     MSDOS            ;INT 21H.
        ENDM

;**********************************************************
;This macro will close a file and update the directory.
;DS:DX must point to an open FCB.
;On return, AL = 00H then entry was found in the directory.
;           AL = 0FFH then entry was not found in the directory
;Function 10H

;
@CLOSE  MACRO   FCB
        MOV     AH,F_CLOSE
        LEA     DX,FCB
        INT     MSDOS
        ENDM
;**********************************************************
;This macro will search a directory for the first entry
;which matches the file name specified in the FCB.
;If it is found, an unopened FCB of
;the same type will be created at the disk transfer address.
;See the IBM Technical Reference Manual for a description
;of this function.
;DS:DX Must point to an unopened FCB of the file to
;      search the directory for.
;On exit AL = 00H if the directory entry was found.
;        AL = 0FFH if the directory entry was not found.

;
;Function 11H
;
@DIR_SEARCH     MACRO   FCB
        LEA     DX,FCB           ;POINT TO THE FCB
        MOV     AH,F_DIR_SEARCH  ;MSDOS FUNCTION CODE
        INT     MSDOS            ;SYSTEM CALL
        ENDM
;**********************************************************
;This macro will invoke the MSDOS function to search the
;directory for the next entry which matches the file name
;specified in the FCB. It is normally used after the
;@DIR_SEARCH. If the file name specified in the unopened FCB
;matches an entry in the directory, an unopened FCB is created
;at the DTA.
;If on return AL = 00H then the entry was found in the directory.
;If AL = 0FFH, then the entry was not found.
;See page 5-23 of the IBM Technical Manual for a complete
;description of this function.
;Function 12H
@SEARCH_NEXT    MACRO   FCB
        LEA     DX,FCB
        MOV     AH,F_SEARCH_NEXT          ;DOS FUNCTION CODE
        INT     MSDOS                     ;SYS. CALL
        ENDM
;**********************************************************
;This macro will invoke the MSDOS function which will
;delete all entries matching the file spec contained in the
;FCB pointed to by DS:DX.
;On return AL = 00H then entry found.
;If AL = 0FFH then the entry was not found.
;See IBM Technical Reference Manual page 5-23.
;Function 13H
```

Listing 6-1 continued

```
@DELETE_FILES   MACRO   FCB
        LEA     DX,FCB
        MOV     AH,F_DELETE_FILES
        INT     MSDOS
        ENDM
;**********************************************************
;This macro will read a sequential record. The record read
;will be the one specified by the current block (byte 0CH of
;the FCB, and the current record (byte 20H of the FCB). The
;record is transferred to the DTA. The record size is specified
;by the value contained in byte 0EH of the FCB.
;DS:DX must point to the FCB on entry to this function.
;On return, AL = 00H then read was successful.
;                01H then EOF was encountered; No data in record.
;                02H DTA contained insufficient space to
;                    read a record. Operation canceled.
;                03H EOF encountered, partial record read,
;                    record padded with zeros.
;Function 14H
@READ_SRECORD   MACRO   FCB
        LEA     DX,FCB          ;POINT TO THE FCB.
        MOV     AH,F_READ_SRECORD ;MSDOS FUNCTION CODE
        INT     MSDOS           ;SYS. CALL
        ENDM
;*************************************************************
;This macro will write a sequential record to disk.
;On entry, the opened FCB must be pointed to by DS:DX
;on exit, AL = 00H write was successful.
;              01H disk was full, operation canceled.
;              02H there was not enough room in the DTA
;                  operation canceled.
;Function 15H
@WRITE_SRECORD  MACRO   FCB
        LEA     DX,FCB          ;POINT TO FCB
        MOV     AH,F_WRITE_SRECORD ;WRITE FUNCTION CODE
        INT     MSDOS
        ENDM
;********************************************************
;This macro will invoke the MSDOS function to Create a file
;or it will erase the contents of an existing file and
;the FCB is opened for the new file.
;DS:DX must point to the unopened FCB.
;On return, AL = 00H then an empty directory entry was found.
;                0FFH then an empty directory entry was not found.
;Function 16H
;
@CREATE_FILE    MACRO   FCB
        LEA     DX,FCB          ;MOVE FCB TO DX
        MOV     AH,F_CREATE_FILE ;MSDOS FUNCTION CODE
        INT     MSDOS            ;SYS. CALL
        ENDM
;***********************************************************
;This macro will invoke the MSDOS function to rename an existing
;file. The current drive and filename are located in the FCB
;at bytes 00 = current disk drive.
;         01-08 = file name
;         09-0BH = file name extension
;DS:DX+11H = START OF NEW FILENAME.
;On return, AL = 00H = Success
```

```
;                 0FFH = No directory entry or match found.
;Function 17H
@RENAME_FILE    MACRO   FCB
        LEA     DX,FCB
        MOV     AH,F_RENAME_FILE        ;FUNCTION CODE
        INT     MSDOS
        ENDM
;*********************************************************
;This macro will fetch the code of the currently selected
;disk drive. The code is returned in AL as 00 = 'A', 01 = 'B'
;etc.
;Function 19H
@CURRENT_DISK   MACRO
        MOV     AH,F_CURRENT_DISK       ;MSDOS FUNCTION CODE
        INT     MSDOS                   ;
        ENDM
;*********************************************************
;This macro set the disk transfer address to the address
;supplied in DS:DX.
;If this function is not set then the default DTA at 80H
;in the PSP is used.
;Function 1AH
@SET_DTA        MACRO   DTA
        LEA     DX,DTA          ;MOVE ADDRESS OF DTA TO DX
        MOV     AH,F_SET_DTA
        INT     MSDOS
        ENDM
;*********************************************************
;This macro will invoke the MSDOS function to read
;the File Allocation Table information. On return, DS:BX
;points to the FAT byte containing the default drive.
;DX contains the number of allocation units
;AL contains the number of sectors per allocation unit, and
;CX contains the size of the physical sector. This is only
;true for MSDOS version 1.0. For MSDOS 2.0 and 2.1 use function
;36H
;
;Function 1BH
@GET_FAT        MACRO
        MOV     AH,F_GET_FAT
        INT     MSDOS
        ENDM
;*********************************************************
;This macro will invoke the random record read function of
;MSDOS. DS:DX must point to an opened FCB for the file to
;be read.
;On return AL = 00H then read was successful
;               01H then EOF, no data returned.
;               02H then DTA contained insufficient space
;                   to read a record. Operation aborted.
;               03H then EOF encountered, partial record
;                   read and padded with zeros.
;Function 21H
@READ_RRECORD   MACRO   FCB
        LEA     DX,FCB
        MOV     AH,F_READ_RRECORD
        INT     MSDOS
        ENDM
;*********************************************************
;This macro will invoke a function to write a random record
```

Listing 6-1 continued

```
;to disk. The current block (FCB + 0CH), current record
;(FCB + 20H) are set according to the relative record field
;(FCB + 21H). The record written is contained in the DTA.
;See page 5-26 of the IBM technical reference manual for
;further information.
;
;Function 22H
@WRITE_RRECORD  MACRO   FCB
        LEA     DX,FCB          ;MOVE FCB INTO DX
        MOV     AH,F_WRITE_RRECORD
        INT     MSDOS
        ENDM
;************************************************************
;This function will invoke the MSDOS function to determine
;the number of records in a file. The filename in contained
;in an unopened FCB. YOU MUST SET THE FCB RECORD SIZE FIELD
; (FCB + 0EH) BEFORE USING THIS FUNCTION!
;The FCB must be unopened.
;On return AL = 00H means the directory entry was found.
;          AL = 0FFH means the directory entry was not found.
;Function 23H
;
@GET_FILE_SIZE  MACRO   FCB
        LEA     AH,F_GET_FILE_SIZE
        MOV     DX,FCB
        INT     MSDOS
        ENDM
;************************************************************
;This macro will set the relative record for random file access.
;The relative record field (FCB + 21H) is set to the same
;address as the current block (FCB + 0CH), and current record
;field (FCB + 20H).
;DS:DX points to an open FCB
;Function 24H
;
@SET_REL_RECORD MACRO   FCB
        LEA     DX,FCB
        MOV     AH,F_SET_REL_RECORD
        INT     MSDOS
        ENDM
;**********************************************************
;This macro will set  a new interrupt vector for the
;specified type.
;AL = TYPE number of interrupt to set vector for.
;DS:DX = 4 byte interrupt handler address.
;
;Function 25H
;
@SET_INT_VECTOR MACRO   TYPE,SEG_VECTOR,OFFSET_VECTOR
        PUSH    DS      ;SAVE OLD DATA SEG.
        MOV     AX,SEG_VECTOR   ;GET SEGMENT ADDRESS
        MOV     DS,AX           ;PUT INTO DS
        MOV     DX,OFFSET_VECTOR ;GET OFFSET INTO SEGMENT
        MOV     AL,TYPE
        MOV     AH,F_SET_INT_VECTOR ;MSDOS FUNCTION CODE
        INT     MSDOS
        POP     DS      ;RESTORE OLD DATA SEG.
        ENDM
;**********************************************************
```

```
;This macro will perform a random block read.
;CX contains the number of records to be read
;DS:DX point to the open FCB.
;On return CX = number of records read.
;       If AL = 00H then block read was successful.
;               01H then EOF encountered, last record intact.
;               02H address wrap above FFFFH in DTA
;               03H EOF encountered last record only partially read
;
;Function 27H
@BLOCK_RREAD    MACRO   FCB,REC_COUNT,RECORD_SIZE
        LEA     DX,FCB                          ;Point to FCB
        MOV     CX,REC_COUNT                    ;How Many records?
        MOV     FCB[0EH],RECORD_SIZE            ;And what Size?
        MOV     AH,F_BLOCK_RREAD
        INT     MSDOS
        ENDM
;*********************************************************
;This macro invokes the random block write function
;under MSDOS control. See IBM technical reference manual
;Page 5-29 for a description of this function.
;CX = number of records to write
;If CX is zero on entry, then file size is set according to
;the random record field and the logical record size.
;DS:DX = pointer to FCB
;On return, AL = 00H block write was successful
;                01H = no records written because disk is full.
;
;Function 28H
;
@BLOCK_RWRITE   MACRO   FCB,REC_COUNT,RECORD_SIZE
        LEA     DX,FCB
        MOV     CX,REC_COUNT
        MOV     FCB[0EH],RECORD_SIZE
        MOV     AH,F_BLOCK_RWRITE
        INT     MSDOS
        ENDM
;*************************************************************
;This function will parse a string for a specified filename.
;See page 5-28 of the IBM technical reference manual for a
;description of this function.
;
;DS:SI point to the command line to parse
;ES:DI point to a portion of memory to be used as the FCB
;Bits 0-3 of AL are used to control the parsing.
; BIT 0 = 0 = Stop parsing if a file separator encountered.
;       = 1 = Ignore file separators.
; BIT 1 = 0 = Set drive in the FCB to default drive (0)
;             if the string does not contain a drive number.
;         1 = Default Drive is not changed if the string
;             does not contain a drive number.
; BIT 2 = 0 = Do not change the filename in the FCB if the
;             string does not contain a name.
;         1 = Change file name in FCB if string contains
;             a file name.
; BIT 3 = 0 = The file extension in the FCB is not changed
;             if the string does not contain an extension.
;         1 = Change the extension if the command line contains
;             an extension.
;
```

Listing 6-1 continued

```
;Function 29H
;
@PARSE_STRING   MACRO   NAME_STRING,FCB_AREA,PARSE_BITS
        LEA     SI,NAME_STRING  ;STRING TO PARSE
        LEA     DI,FCB_AREA
        PUSH    ES              ;SAVE SEG REGISTERS ES AND DS
        PUSH    DS
        POP     ES              ;SET ES = DS
        MOV     AL,PARSE_BITS   ;HOW DO WE PARSE?
        MOV     AH,F_PARSE_STRING ;
        INT     MSDOS
        POP     ES              ;Restore seg register
        ENDM
;**********************************************************
;This macro will invoke the MSDOS function to read the
;date and return the date in the following registers;
;CX: = Year in binary
;DH: = Month in binary (1 = Jan. etc.)
;DL: = Day of the month in binary
;
@GET_DATE       MACRO
        MOV     AH,F_GET_DATE
        INT     MSDOS
        ENDM
;**********************************************************
;This macro will set the date according to the binary information
;contained in the registers as follows.
;CX: = Year
;DH: = Month (1 = Jan. etc.)
;DL: = Day of the month
;
;On return AL = 00H then the date code was valid.
;          AL = 0FFH then the date was not valid, operation
;               aborted.
;Function 2BH
;
@SET_DATE       MACRO   YEAR,MONTH,DAY
        MOV     AH,F_SET_DATE   ;GET MSDOS FUNCTION CODE
        MOV     CX,YEAR         ;LOAD YEAR IN CX
        MOV     DH,MONTH        ;MONTH IN DH
        MOV     DL,DAY
        INT     MSDOS
        ENDM
;**********************************************************
;This macro will get the current time and return it in the
;registers as depicted below.
;CH: = Hour (24 hour format)
;CL: = minutes
;DH: = Seconds
;DL: = hundreths of a second
;
;Function 2CH
;
@GET_TIME       MACRO
        MOV     AH,F_GET_TIME
        INT     MSDOS
        ENDM
;**********************************************************
;This macro will invoke the MSDOS function to set the time
```

```
;The following registers are used to specify the time on
;invoking this function.
;CH: = HOUR (24 HOUR FORMAT)
;CL: = MINUTES
;DH: = SECONDS
;DL: = HUNDRETHS OF A SECOND
;
;FORMAT 2DH
;
@SET_TIME       MACRO   HOUR,MINUTES,SECONDS,HUNDRETHS_SEC
        MOV     AH,F_SET_TIME
        MOV     CH,HOUR
        MOV     CL,MINUTES
        MOV     DH,SECONDS
        MOV     DL,HUNDRETHS_SEC
        INT     MSDOS
        ENDM
;*********************************************************
;This macro will set the verify flag used when writing data
;to disk. When on, it causes a verify operation after each
;write. On entry AL = 00H then do not verify.
;                     01H then verify.
;Function 2EH
;
@VERIFY_WRITE   MACRO ON_OR_OFF
        MOV     AH,F_VERIFY
        MOV     AL,ON_OR_OFF
        INT     MSDOS
        ENDM
;********************************************
;This macro will return a pointer to the DTA.
;ES:BX will contain the pointer on return.
;
;Function 2FH
;
@GET_DTA        MACRO
        MOV     AH,F_GET_DTA
        INT     MSDOS
        ENDM
;**********************************************
;This macro will get the version number of MSDOS.
;On return AL contains the major revision level, and AH
;will contain the minor rev. number.
;
;Function 30H
;
@GET_VERSION    MACRO
        MOV     AH,F_GET_VERSION
        INT     MSDOS
        ENDM
;*********************************************************
;This macro will invoke the MSDOS terminate/keep resident
;function. See page 5-31 of the IBM DOS Technical Reference
;Manual. On return, DX contains the memory size in paragraphs.
;AL = exit code.
;
;Function 31H
;
@TERMINATE_RESIDENT     MACRO   MEM_SIZE,EXIT_CODE
        MOV     AL,EXIT_CODE
```

Listing 6-1 continued

```
       MOV     DX,MEM_SIZE
       MOV     AH,F_TERMINATE_RESIDENT
       INT     MSDOS
       ENDM
;******************************************************
;This macro will invoke the MSDOS function to either
;read the current state of ^C (Control C) checking
;(ON/OFF) or to set the state of ^C checking. If the state
;is set to ON, then functions 06H and 07H cannot be used.
;If AL = 00 then the current state is returned in DL which is:
;00 = off, 01H = ON.
;If AL =01 then set the state as defined by the contents of DL
;(00H = Set ^C checking Off which is the default state)
;(01H = Set ^C checking On)
;Use this function to enable Control C checking during all operations.
;
;Function 33H
;
@SET_C_CHECK   MACRO   SWITCH1,SWITCH2
       LOCAL   FETCH
       MOV     AL,SWITCH1                     ;SET OR FETCH?
       OR      AL                             ;SET FLAGS
       JZ      FETCH                          ;Z, THEN GET CURRENT STATE
       MOV     DL,SWITCH2                     ;ELSE LOAD DL WITH ON OR
                                              ;OFF.
FETCH: MOV     AH,F_SET_C_CHECK                      ;MSDOS FUNCTION CODE
       INT     MSDOS
       ENDM
;******************************************************
;This macro will return an interrupt vector for a given
;interrupt type. Use this call to retrieve a vector and
;function call 25H to set a vector. The old vector read by
;this function should be saved and restored prior to exiting
;a user program and returning to DOS.
;AL contains the interrupt number on entry.
;And ES:BX contain the vector on return.
;
;Function 35H
;
@READ_VECTOR   MACRO   TYPE
       MOV     AL,TYPE
       MOV     AH,F_READ_VECTOR
       INT     MSDOS
       ENDM
;********************************************************
;This macro will invoke the MSDOS function which returns
;the amount of free space remaining on a disk (in BX).
;It also returns the total number of allocation units on the
;drive(in DX), bytes per sector (in CX), Sectors per allocation
;unit (in AX).
;If AX contains 0FFFFH on return then the drive number was
;invalid. This call is to be used instead of function 1BH for
;DOS 2.0.
;
;Function 36H
;
@GET_FREE_SPACE MACRO   DRIVE
       MOV     DL,DRIVE        ;GET DRIVE NUMBER (0=A ETC.)
       MOV     AH,F_GET_FREE_SPACE
```

```
        INT     MSDOS
        ENDM
;**********************************************************
;This macro will create a sub-directory at the end of a
;user supplied path name.
;DS:DX must point to an ASCII string terminated with a null
;00H byte (ASCIIZ string). The string must contain drive and
;pathname.
;On return if the carry is set, AX contains an error code:
;If AX = 03H then the pathname was not found.
;If AX = 05H then access was denied to the parent directory.
;
;Function 39H
;
@MK_DIR MACRO   PATH_NAME
        LEA     DX,PATH_NAME
        MOV     AH,F_MK_DIR
        INT     MSDOS
        ENDM
;**********************************************************
;This macro will remove a directory entry from the parent.
;DS:DX point to the ASCIIZ pathname.
;On return if the carry is set, there was an error. The
;error value in AX is the same as for function 39H above,
;with the addition of 10H which would mean that the path specified
;was the current directory of on a drive.
;
;Function 3AH
;
@RM_DIR MACRO   PATH_NAME
        LEA     DX,PATH_NAME
        MOV     AH,F_RM_DIR
        INT     MSDOS
        ENDM
;**********************************************************
;This function will change the current directory.
;DS:DX points to an ASCIIZ string which contains the new
;directory name. If on return the carry flag is set,
;AX will contain an error code. 03H means that the pathname was
;not found, or was not valid.
;
;Function 3BH
;
@CH_DIR MACRO   PATH_NAME
        LEA     DX,PATH_NAME
        MOV     AH,F_CH_DIR
        INT     MSDOS
        ENDM
;**********************************************************
;This macro will invoke a function to create a file.
;The file is created if not found in the directory,
;or set to a null file (sets length to zero) if the entry
;already exists in the directory.
;DS:DX point to the ASCIIZ path name and CX contains
;the file attributes:
;01 = set = read only
;02 = set = hidden file
;04 = set = system file
;08 = set = entry contains the volume label
;10H = set = entry defines sub-directory
```

Listing 6-1 continued

```
;20H = set = set whenever the file has been written to and closed.
;File attributes can be combined.
;On return if the carry is set, AX will contain an error code
;as follows: 03 = path name not found (invalid)
;            04 = Too many open files.
;            05 = Access denied
;
;If the carry is not set, the operation was successful, and
;AX will contain a file handle number which is required
;For future file access functions.
;
;
;Function 3CH
;
@CREATE_FILE_2  MACRO   PATH_NAME,ATTRIBUTES
        LEA     DX,PATH_NAME
        MOV     CX,ATTRIBUTES
        MOV     AH,F_CREATE_FILE_2
        INT     MSDOS
        ENDM
;*********************************************************
;This macro will open a file. AL contains the desired
;access code as follows:
;00 = file is to be opened as read only.
;01 = the file is opened as write only.
;02 = the file is opened as read/write.
;
;DS:DX points to an ASCIIZ pathname.
;The record size is initialized to 1 byte, and the R/W pointer is set
;to the first byte of the file.
;On return if the carry is set, AX contains an error code as
;follows: 02 = File not found
;         04 = Too many open files
;         05 = Access denied
;         12 = Invalid Access (Code in AL not between 0 -2)
;If the operation was successful, then AX will contain a
;16 bit file handle which must be used to access the file
;during subsequent operations.
@OPEN_FILE_2    MACRO   PATH_NAME,ACCESS
        LEA     DX,PATH_NAME
        MOV     AL,ACCESS
        MOV     AH,F_OPEN_FILE_2
        INT     MSDOS
        ENDM
;*********************************************************
;This macro will close a file.
;BX must contain the file handle number of the file to be
;closed.
;On return, if the carry is set there was an error and the
;error code 06 will be in AX which indicates the file handle
;was not for an open file.
;
;Function 3EH
;
@CLOSE_FILE_2   MACRO   HANDLE
        MOV     BX,HANDLE
        MOV     AH,F_CLOSE_FILE_2
        INT     MSDOS
        ENDM
```

```
;**************************************************************
;This macro will read from a file or device.
;DS:DX points to a buffer for the operation.
;CX = the number of characters to read.
;BX = file handle number
;Not all characters specified in CX need be read. See page 5-35
;of the IBM DOS technical reference manual.
;this function.
;
;On exit if the carry flag is set, then AX will contain
;an error code as follows:
;05 = Access denied.
;06 = Invalid Handle
;If the carry is not set then AX will equal the number of bytes
;read from the file or device.
;
;Function 3FH
;
@READ_FILE_2    MACRO   HANDLE,BUFFER,BYTES
        LEA     DX,BUFFER
        MOV     BX,HANDLE
        MOV     CX,BYTES
        MOV     AH,F_READ_FILE_2
        INT     MSDOS
        ENDM
;***************************************************************
;This macro will write a specified number of bytes to a file
;or to a device. DS:DX must point to the buffer from which the
;data is transferred, CX contains the total number of bytes to
;write to the file, and BX must contain the file handle number.
;On return, if the carry is set then AX will contain an error code
;number as in the read function above (function # 3FH).
;
;If the carry is not set then AX equals the number of bytes written
;to the file or device.
;
;Function 40H
;
@WRITE_FILE_2   MACRO   BUFFER,HANDLE,BYTES
        LEA     DX,BUFFER
        MOV     AH,F_WRITE_FILE_2
        MOV     CX,BYTES
        MOV     BX,HANDLE
        INT     MSDOS
        ENDM
;********************************************************************
;This macro will invoke the MSDOS function to delete a directory
;entry. DS:DX must point to the pathname of the entry to delete.
;If the carry is set on return, there was an error. AX will contain
;one of the following error codes: 02 = File not found.
;                                  05 = Access denied.
;This function only will work on files that are closed, and
;have the proper attributes (i.e. R/W).
;
;Function 41H
;
@DELETE_FILE_2  MACRO   PATH_NAME
        LEA     DX,PATH_NAME
```

Listing 6-1 continued

```
        MOV     AH,F_DELETE_FILE_2
        INT     MSDOS
        ENDM
;*****************************************************
;This macro will move the pointer which is used
;for read and write operations into the specified file
;by 1 of 3 methods.
;If AL = 0 then the pointer will be moved the specified
;          number of bytes (in CX:DX) from the beginning of
;          the file (absolute seek).
;If AL = 1 then the file pointer will be moved the
;          specified number of bytes (in CX:DX) from
;          the current pointer position.
;If AL = 2 then the file pointer will be moved the
;          specified number of bytes (in CX:DX) from
;          the end of the file. Use an offset of zero
;          to determine the end of the file.
;CX and DX must contain the distance to move the pointer in bytes.
;          (CX = MSW, DX = LSW).
;AL = type of move.
;BX = file handle.
;On return if the carry is not set, then the operation was
;successful and DX:AX contains the new pointer position.
;If the carry is set, then AX contains an error code as follows:
;AX = 01 then the value in AL was not in the range of 0-2.
;AX = 06 then the handle was invalid (an unopen file).
;
;Function 42H
;
@MOVE_POINTER   MACRO   MSW_DISTANCE,LSW_DISTANCE,TYPE,HANDLE
        MOV     CX,MSW_DISTANCE ;HIGH ORDER WORD FOR DISTANCE
        MOV     DX,LSW_DISTANCE ;LOW ORDER WORD FOR DISTANCE
        MOV     AL,TYPE         ;HOW WE MOVE THE POINTER
        MOV     BX,HANDLE       ;HANDLE NUMBER OF OPEN FILE.
        MOV     AH,F_MOVE_POINTER
        INT     MSDOS
        ENDM
;*********************************************************
;This macro will invoke the MSDOS function to either read
;a file's attributes or change them as defined by the contents
;of CX.
;AL contains 00H to read the file attributes, or
;01H to set the attributes.
;DS:DX must point to the files path name.
;If the attributes are read, then they are returned in
;the CX register (if there were no errors during the operation).
;If an error occurred, the carry bit will be set and AX will
;contain one of the following error codes.
;If AX = 1 then the value in AL on entry was not 00 or 01H
;   AX = 3 then the path name was invalid.
;   AX = 5 then access was denied.
;
;Function 43H
;
@CHANGE_ATTRIBUTES      MACRO   PATH_NAME,ATTRIBUTES,TYPE
        LEA     DX,PATH_NAME
        MOV     CX,ATTRIBUTES
        MOV     AH,F_CHANGE_ATTRIBUTES
        MOV     AL,TYPE
        INT     MSDOS
        ENDM
;**********************************************************
;This macro will duplicate a file handle of an open file.
;BX = file handle number to be duplicated
;
;Function 45H
@DUP_HANDLE     MACRO   HANDLE
        MOV     BX,HANDLE
        MOV     AH,F_DUP_HANDLE
        INT     MSDOS
        ENDM
;**********************************************************
;This macro will point an existing file handle to a new file.
;CX = new file handle number
;BX = existing file handle number
;On return:
;If the carry is set then AX contains an error code as follows:
;04 = Too many files open.
;06 = Invalid handle.
;If the carry is not set, both handles will point to the same
;(new) file.
;
;Function 46H
;
@DUP_HANDLE_2   MACRO   OLD_HANDLE,NEW_HANDLE
        MOV     CX,NEW_HANDLE
        MOV     BX,OLD_HANDLE
        MOV     AH,F_DUP_HANDLE_2
        INT     MSDOS
        ENDM
;**********************************************************
;This macro will get the current directory (the full path)
;and read it into a 64 byte buffer pointed to by DS:SI.
;DL must contain the drive number (0=A etc.)
;On return DS:DI will point to the buffer containing the
;directory.
;If the carry is set then an error occurred in that the drive
;specified in DL was invalid (AX = 15H)
;
;Function 47H
;
@CURRENT_DIR    MACRO   BUFFER,DRIVE_NUM
        MOV     DL,DRIVE_NUM
        LEA     SI,BUFFER
        MOV     AH,F_CURRENT_DIR
        INT     MSDOS
        ENDM
;**********************************************************
;This macro will invoke the MSDOS function to return a pointer
;to a free block of memory of a specified size.
;(Almost sounds like multitasking!)
;BX contains the amount of memory to be allocated (in paragraphs).
;On return the carry will be set if there was an error and
;AX contains 07 then the memory control blocks were destroyed.
;                  08 = Requested size larger than available RAM.
;BX will contain the maximum size that was allocated (in paragraphs).
;If the carry is not set on return, then the operation was
;successful, and AX contains the pointer to the allocated block.
;
```

Listing 6-1 continued

```
;Function 48H
;
@ALLOCATE_MEM   MACRO   BLOCK_SIZE
        MOV     BX,BLOCK_SIZE
        MOV     AH,F_ALLOCATE_MEM
        INT     MSDOS
        ENDM
;*********************************************************
;This macro will return (deallocate) a block of memory
;which was previously allocated.
;On entry, ES must contain the segment address of the block to
;be returned.
;On returning from the routine, if the carry is set then
;AX will contain one of the following error codes:
;AX = 07 then memory control blocks were destroyed.
;AX = 09 then the block contained in ES was not one
;        allocated through the ALLOCATE_MEM (48H) function.
;If the carry bit is not set, then the operation was successful.
;
;Function 49H
@DEALLOCATE_MEM MACRO   BLOCK_NUM
        MOV     ES,BLOCK_NUM
        MOV     AH,F_DEALLOCATE_MEM
        INT     MSDOS
        ENDM
;*********************************************************
;This macro will invoke the MSDOS function which modifies
;an allocated block of memory. ES must contain the segment of
;block to modify, and BX contains the new block size (in paragraphs).
;If the new size of the block is less than that which is allocated,
;then MSDOS will attempt to reduce the size of the block. If the
;requested block size is larger than that which is already
;allocated then MSDOS will attempt to enlarge the amount of
;memory allocated. Microsoft refers to this process as
;"shrinking", or "growing" allocated memory blocks.
;On return of the carry is set then AX will contain an error
;code as follows:
;AX = 07H then Memory Control Blocks were destroyed.
;      08H then there was not enough memory to enlarge the block.
;      09H then the block specified in ES was invalid.
;BX will also contain the maximum memory (in paragraphs) that
;is available if the specified block size was unavailable (AX = 07).
;
;If the carry is not set, then the operation was successful.
;
;Function 4AH
;
@NEW_BLK_SIZE   MACRO   NEW_SIZE,OLD_BLOCK
        MOV     ES,OLD_BLOCK
        MOV     BX,NEW_SIZE
        MOV     AH,F_NEW_BLK_SIZE
        INT     MSDOS
        ENDM
;************************************************************
;This macro will load and/or execute a program.
;On entry DS:DX will point to the pathname of the file to load
;and/or execute. The pathname must be in ASCII and terminated
;with a null byte (00H). ES:BX must point to a parameter block
;(See IBM Technical Reference Manual page 5-43). AL must contain
```

```
;the function code (00 = load and execute the program,
;03H = load the program, do not begin execution.
;Again, see the IBM DOS Technical Reference Manual for a complete
;description of this function.
;
;Function 4BH
;
@LOAD_EXECUTE   MACRO   PATH_NAME,PARAM_BLK,FUNCTION
        LEA     DX,PATH_NAME
        LEA     BX,PARAM_BLK
        MOV     AL,FUNCTION
        MOV     AH,F_LOAD_EXECUTE
        INT     MSDOS
        ENDM
;*****************************************************************
;This macro will terminate the current process. Control is returned
;to the process which invoke the terminated one. AL is used to
;return a code to the parent process.
;This operation will also close any and all open files.
;
;Function 4CH
;
@EXIT   MACRO   RETURN_CODE
        MOV     AL,RETURN_CODE
        MOV     AH,F_EXIT
        INT     MSDOS
        ENDM
;*****************************************************************
;This macro will retrieve the return code of the child process
;which terminated via the @EXIT function (4CH).
;A non-multitasking child process which is executed using the
;@LOAD_EXECUTE function (4BH), will terminate using the
;@EXIT function (4CH). The @EXIT function returns a one byte
;return code on execution. This code can be used to convey
;status information from the child process to the parent process.
;Therefore, the parent program should use the @GET_RET_CODE function
;to retrieve this status byte passed from the child process.
;This function can only be used once to obtain the return code.
;On return, AL will contain the return code, and AH will contain
;another code as follows:
;AH = 00 = Child process terminated/aborted (normal return)
;     01 = A control C caused process termination.
;     02 = A critical error has occurred.
;     03 = If the child process was terminated via
;          the @TERMINATE_RESIDENT function (31H).
;Function 4DH
;
@GET_RET_CODE   MACRO
        MOV     AH,F_GET_RET_CODE
        INT     MSDOS
        ENDM
;*****************************************************************
;This macro will invoke the MSDOS function which will find the
;first matching pathname (drive/path/filename) which has the
;attributes specified in the CX register. This function is
;described on page 5-46 of the IBM DOS Technical Reference Manual.
;The function will create a data block of information about the file
;in the DTA. DS:DX must point to the pathname on entry.
;On return, if the carry is set then AX contains an error code
;as follows:
```

Listing 6-1 continued

```
;AX = 02 then the pathname was invalid.
;AX = 18 then there were no files found.
;
;Function 4EH
;
@FIND_FILE      MACRO   PATH_NAME,ATTRIBUTES
        MOV     CX,ATTRIBUTES
        LEA     DX,PATH_NAME
        INT     MSDOS
        ENDM
;**************************************************************
;
;**************************************************************
;This macro will find the next file with matching pathname
;and attributes. It is used after the @FIND_FILE function (4EH).
;If the carry is set and AX contains 18H on return
;then there were no more files found.
;
;Function 4FH
;
@FIND_NEXT_FILE MACRO
        MOV     AH,F_FIND_NEXT_FILE
        INT     MSDOS
        ENDM
;**************************************************************
;This macro will read the verify flag and return it's status in
;AL. If AL = 00H then the verify flag is off.
;If AL = 01H then the verify flag is on.
;
;Function 54H
;
@READ_VERIFY_FLAG       MACRO
        MOV     AH,F_READ_VERIFY_FLAG
        INT     MSDOS
        ENDM
;**************************************************************
;This macro will invoke the MSDOS function to rename a directory
;entry. DS:DX must point to an ASCIIZ string which contains the
;pathname of the file to be renamed. ES:DI points to the new
;pathname of the file. The disk specification must be the same
;In both the old and new pathnames.
;On return if the carry is set then an occurred and AX contains
;the error code as follows:
;If AX = 3 then the pathname specified by DS:DX was not found.
;If AX = 5 then access was denied either to the source or
;         destination file.
;If AX = 17 then the source and destination pathnames exist
;          on different drives.
;
;Function 56H
;
@RENAME_FILE2   MACRO   SOURCE,DESTINATION
        LEA     DX,SOURCE
        LEA     DI,DESTINATION
        MOV     AH,F_RENAME_FILE2
        INT     MSDOS
        ENDM
;**************************************************************
;This macro will invoke the MSDOS function to read or set the date
```

```
;and time stamp of the last write for a specified file.
;The file's handle must be in BX on entry, and the desired
;function code in AL (00H = get date and time, 01H = set date
;and time). CX will contain the time to be set and DX will
;contain the date to be set.
;If the function specified was to read the date and time,
;then CX will contain the time and DX will contain the date
;of the last write operation, on return from the function.
;If the carry is set on return then AX will contain an error code
;as follows:
;AX = 01 then the argument in AL on entry was invalid, it was
;        not  00 or 01H.
;AX = 06 then the file handle passed in BX was invalid.
;
;Function 57H
;
@GET_SET_DT     MACRO   FUNCTION,HANDLE,TIME,DATE
        MOV     AL,FUNCTION
        MOV     BX,HANDLE
        MOV     CX,TIME
        MOV     CX,TIME
        MOV     DX,DATE
        MOV     AH,F_GET_SET_DT
        INT     MSDOS
        ENDM
;*****************************************************************
;****************************************************************
;
;********    VIDEO DISPLAY MACROS    *********************
;********    USING BIOS CALLS FOR VIDEO CONTROL **********
;******** BIOS INTERRUPT TYPE: 10H
;
;POSSIBLE ATTRIBUTE BYTES AS DEFINED IN IBM TECHNICAL
;REFERENCE MANUAL ARE:
;B7  B6  B5  B4  B3  B2  B1  B0
;================================  Color Text mode:
;B   R   G   B   I   R   G   B     B0-B3 = FOREGROUND
;L   E   R   L   N   E   R   L     B4-B6 = BACGROUND
;I   D   E   U   T   D   E   U     0 = BLACK 1 = BLUE
;N       E   E   E       E   E     2 = GREEN   3= CYAN
;K       N       N       N         4 = RED     5= MAGNETA
;                S                 6 = BROWN   7= WHITE
;                I                 8 = GRAY    9= LIGHT BLUE
;                T                 10 = LIGHT GREEN 11= LIGHT
;                Y                                   CYAN
;                                  12= LIGHT RED 13= LIGHT
;                                                 MAGNETA
;                                  14= YELLOW    15= WHITE
;                                                 HIGH INTENSITY
;
;MONOCHROME EXAMPLES:
;0   0   0   0   0   0   0   0    = NO DISPLAY BLACK/BLACK
;0   0   0   0   0   0   0   1    = UNDERLINE CHARACTER
;0   0   0   0   0   1   1   1    = NORMAL WHITE/BLACK
;0   1   1   1   0   1   1   1    = NO DISPLAY WHITE/WHITE
;1   1   1   1   1   0   0   0    = BLINK, REVERSE VIDEO,HIGH
;                                   INTENSITY
;
;COLOR CHARACTER EXAMPLE:
;0   1   0   1   1   0   0   1    = Magneta background,
;                                   Character is Light blue
```

Listing 6-1 continued

```
;**************************************************************
;THIS MACRO WILL SET THE VIDEO MODE.  PASS THE ARGUMENT AS
;FOLLOWS: AL= 0 = TEXT 40 X 25-B/W
;         AL= 1 = TEXT 40 X 25-COLOR
;         AL= 2 = TEXT 80 X 25-B/W
;         AL= 3 = TEXT 80 X 25-COLOR
;         AL= 4 = GRAPHICS, 320 X 200-B/W
;         AL= 5 = GRAPHICS, 320 X 200-COLOR
;         AL= 6 = GRAPHICS, 640 X 200-B/W
;
;THIS IS A BIOS CALL AND AS SUCH MAY NOT WORK WITH
;ALL PC "COMPATIBLES".
;
;Function 00H
;
@VDMODE   MACRO     MODE
          MOV  AH,F_MODESET    ;MODE SET
          MOV  AL,MODE
          INT  BIOS            ;BIOS CALL
          ENDM
;*********************************************************
;THIS MACRO ALLOWS YOU TO CHANGE THE CURSOR.
;LOAD THE VALUE OF THE START SCAN LINE INTO CH
;AND THE END SCAN LINE INTO CL.
;VALID PARAMETERS ARE: 0-13 OR 0-7 FOR COLOR MONITOR
;
;Function 01H
@CURCHG   MACRO     SLINE,ELINE
          MOV  CH,SLINE             ;GET START LINE
          MOV  CL,ELINE             ;GET END LINE
          MOV  AH,F_CURCHG          ;CHANGE CURSOR FUNCTION
          INT  BIOS                 ;BIOS CALL
          ENDM
;**************************************************************
;THIS MACRO WILL SET THE CURSOR POSITION
; BH = PAGE, DH= ROW, DL= COLUMN
;  PAGENUM     ROW       COL
;
;Function: 02H
;
@CURSET   MACRO     PAGENUM,ROW,COL
     MOV  BH,PAGENUM           ;GET THE PAGE NUMBER
     MOV  DH,ROW               ;GET ROW (0-24)
     MOV  DL,COL               ;GET COL (0-79)

     MOV  AH,F_CURSET          ;FUNCTION CODE TO SET CURSOR
     INT  BIOS                 ;BIOS CALL
     ENDM
;***********************************************************
;THIS MACRO WILL FETCH THE CURSOR POSITION. ON RETURN
;THE ROW/COL COORDINATES WILL BE IN DX (DH=ROW,DL=COL)
;
;
;Function: 03H
;
@CURRD    MACRO
          MOV  AH,F_CURRD      ;GET FUNCTION CODE TO READ
                               ;POSITION
          INT  BIOS            ;BIOS CALL
          ENDM
```

```
;*****************************************************
;THIS MACRO WILL READ THE LIGHT PEN POSITION
;IF THE SYSTEM IS SO EQUIPPED.
;ON RETURN: AH=0 PEN NOT TRIGGERED
;OTHERWISE: AH=1 AND DH=ROW, DL=COL
;                    CH=SCAN ROW (0-199)
;                    BX=PIXEL COLUMN (0-319 -LOW RES)
;                                    (0-639- HIGH RES)
;Function 04H
;
@PENRD    MACRO
          MOV  AH,F_PENRD     ;READ PEN POSITION
          INT  BIOS           ;BIOS CALL
          ENDM
;*****************************************************
;THIS MACRO WILL SET THE ACTIVE VIDEO PAGE
;
;Function 05H
;
@VDPGSET  MACRO     PAGENUM
     MOV  AH,F_PGSET           ;PAGE SET FUNCTION
     MOV  AL,PAGENUM          ;PARAMETER= PAGE NUMBER
     INT  BIOS                ;BIOS CALL
     ENDM

;*********************************************************
;THIS MACRO WILL SCROLL A CERTAIN NUMBER OF LINES IN A
;GIVEN DIRECTION, FOR A GIVEN SET OF CORORDINATES, WITH A
;SPECIFIED ATTRIBUTE BYTE.
;aN EXAMPLE WOULD BE TO PASS 06,00,18H,4FH,00,00,07 AS
;PARAMETERS TO THE MACRO. THIS WOULD SCROLL THE SCREEN
;UP (06), FOR ALL LINES IN WINDOW (00), FROM THE TOP LEFT
;CORNER (ROW=0, COLUMN= 0), TO THE LOWER RIGHT CORNER
;(ROW = 18h = 24, COL= 4FH = 79), WITH A NORMAL CHARACTER
;ATTRIBUTE (07). tHE EFFECT OF THESE PARAMETERS WOULD BE
;TO CLEAR THE SCREEN.
;THIS IS A BIOS CALL AND MAY NOT BE COMPATIBLE WITH ALL
;PC CLONES.
;
;Function 06H = Scroll up
;         07H = Scroll Down
;
@SCROLL MACRO   DIRECTION,LINES,LRROW,LRCOL,ULROW,ULCOL,ATTRIBUTE
          MOV  AH,DIRECTION        ;REQUEST SCROLL UP/DOWN
          MOV  AL,LINES            ;NUMBER OF LINES  TO SCROLL
                                   ;0 = ALL LINES IN WINDOW
          MOV  DH,LRROW            ;LOWER RIGHT ROW
          MOV  DL,LRCOL            ;LOWER RIGHT COLUMN
          MOV  CH,ULROW            ;UPPER LEFT ROW
          MOV  CL,ULCOL            ;UPPER LEFT COLUMN
          MOV  BH,ATTRIBUTE        ;ATTRIBUTE BYTE
          INT  BIOS                ;BIOS CALL
          ENDM
;*****************************************************
;THIS MACRO WILL RETURN THE CHARACTER WHICH IS AT
;THE CURRENT CURSOR POSITION.
;BH= PAGE NUMBER
;ON RETURN: AL= CHARACTER, BL= ATTRIBUTE
;
;Function: 08H
;
```

Listing 6-1 continued

```
@CHARRD   MACRO     PAGENUM
          MOV  AH,F_CHARRD
          MOV  BH,PAGENUM
          INT  BIOS
          ENDM
;******************************************************
;THIS MACRO WILL SEND A CHARACTER IN AL TO THE VIDEO
;DISPLAY. BH=PAGE CX= NUMBER OF CHARACTERS TO OUTPUT
;         AL= CHARACTER , BL= ATTRIBUTE BYTE
;
;Function: 09H
;
@VDCHAR2  MACRO     PAGENUM, CHARACTERS, ATTRIBUTE
          MOV  BH,PAGENUM
          MOV  BL,ATTRIBUTE
          MOV  CX,CHARACTERS
          MOV  AH,F_VDCHAR2
          INT  BIOS
          ENDM
;******************************************************
;THIS MACRO WILL SEND THE CHARACTER IN AL TO THE
;VIDEO DISPLAY. THE EXISTING VIDEO ATTRIBUTE IS USED
; BH= PAGE NUMBER, CX= NUMBER OF CHARACTERS
;
;Function: 0AH
;
@VDCHAR1  MACRO     PAGENUM, CHARACTERS
          MOV  BH,PAGENUM
          MOV  CX,CHARACTERS
          MOV  AH,F_VDCHAR1
          INT  BIOS
          ENDM

;******************************************************
;THIS MACRO WILL SET EITHER FOREGROUND OR BACKGROUND
;COLORS. THE CHOICE OF COLOR FOR BACKGROUND IS PUT
;INTO BL (0-15); AND TO SELECT ONE OF TWO COLOR
;PALETTES FOR FOREGROUND, SET BL TO 0 = GR, RED, BRN
;                                   1 = CY, MAGNETA, WHITE
;BH= 0 = BACKGROUND
;BH= 1 = FOREGROUND (PALETTE)
;
;Function: 0BH
;
@COLORSET MACRO     GROUND, COLOR
          MOV  BH,GROUND           ;BACK OR FOREGROUND?
          MOV  AH,F_COLORSET       ;FUNCTION CALL NUMBER
          MOV  BL,COLOR            ;GET COLOR OR PALETTE TYPE
          INT  BIOS
          ENDM
;******************************************************
;THIS MACRO WILL SET (TURN ON) A GRAPHICS PIXEL
;GIVEN IT'S POSITION AND COLOR.
;DX = ROW = 0 TO 199, CX = COL = 0 TO 319 (MED RES)
;                                0 TO 639 (HIGH RES)
;See the video definitions at the beginning of the video
;macros.
;Use this function to reset a pixel by specifying a foreground
;color the same as the background color.
```

```
;
;Function: 0CH
;
@SET     MACRO   ROW,COL,COLOR
           MOV       AH,F_SET        ;SET DOT FUNCTION CODE
           MOV       DX,ROW
           MOV       CX,COL
           MOV       AL,COLOR
           INT       BIOS
           ENDM
;*****************************************************
;THIS MACRO WILL TEST A DOT POSITION. THE COLOR IS RETURNED
;IN AL. DX=ROW, CX= COL.
;
;Function: 0DH
;
@POINT     MACRO      ROW, COL
           MOV  DX,ROW
           MOV  CX,COL
           MOV  AH,F_POINT
           INT  BIOS
           ENDM
;*****************************************************
;THIS MACRO WILL OUTPUT A CHARACTER IN AL TO THE VIDEO
;DISPLAY. BH= PAGE NUMBER, BL= COLOR (GRAPHICS MODE),
;          AL=CHARACTER (MUST BE IN AL PRIOR TO
;                       INVOKING THE MACRO).
;
;Function: 0EH
;
@VDCHAR3   MACRO      PAGENUM, COLOR
           MOV  BH,PAGENUM
           MOV  BL,COLOR
           MOV  AH,F_VDCHAR3         ;BIOS FUNCTION NUMBER
           INT  BIOS
           ENDM
;*****************************************************
;THIS MACRO WILL RETURN THE CURRENT VIDEO MODE IN AL.
;ON RETURN AL= MODE, BH= ACTIVE PAGE, AH= ACTIVE COLUMN
;
;Function: 0FH
;
@VDMODERD MACRO
      MOV  AH,F_MODERD          ;READ FUNCTION MODE
      INT  BIOS
      ENDM
;*****************************************************
;These macros provide an interface to the BIOS routines
;which can be used to access the printer.
;BIOS INTERRUPT TYPE: 17H
;*****************************************************
;The first macro will return the printer's status in AH.
;b7 = 1 = Printer busy
;b6 = 1 = Acknowledge
;b5 = 1 = Paper out
;b4 = 1 = Printer selected
;b3 = 1 = I/O Error
;b2 = x = Unused
;b1 = x = unused
;b0 = 1 = Time Out
```

Listing 6-1 continued

```
;See page A-46 of the IBM Technical Reference manual for
;a description of the routines used for printer BIOS I/O.

@PRINTER_STATUS MACRO
        MOV     AH,F_PRINTER_STATUS         ;Get function code
        INT     17H                         ;Go through BIOS
        ENDM
;**********************************************************
;This macro will send a character in AL to the printer.
;After executing the BIOS routine, the status will be returned
;in AH.
@PRINTER_OUT_B  MACRO
        MOV     AH,F_PRINTER_OUT_B          ;BIOS function code
        INT     17H
        ENDM
;**********************************************************
;This macro function will reset and initialize the printer
;port.
@PRINTER_INIT   MACRO
        MOV     AH,F_PRINTER_INIT
        INT     17H
        ENDM
;**********************************************************
;BIOS INTERRUPT TYPE: 14H
;RS-232C communication functions
;**********************************************************
;These macros will handle serial I/O through BIOS calls.
;See page A22 of the IBM Techniacl reference manual for a
;detailed expalnation of the bit assignments used in the
;RS_INIT macro.
@RS232_INIT     MACRO   CONFIG,PORT
        MOV     AL,CONFIG                   ;Get the configuration.
        MOV     DX,PORT                     ;1 OR 2
        MOV     AH,F_RS232_INIT
        INT     14H                         ;BIOS call
        ENDM
;*********************************************************
;This macro will get a character from the serial port.
;No check in the macro is made to see if there is one ready.
;your program should do this before invoking this macro.
;use the RS_STATUS to see if there is a character ready.
;AH will be zero on return, if a character was recieved
;without error, and is in AL. If AH <> 0 then AH = the status of the port.
;
@RS_INPUT       MACRO PORT
        MOV     DX,PORT                     ;Which port?
        MOV     AH,F_RS_INPUT               ;Function code
        INT     14H                         ;BIOS call
        ENDM
;********************************************************
;This macro will send a character via the serial channel.
;The character to be transmitted is in AL on invoking the macro.
;If b7 of AH is set on return, an error occurred and b6-b0 = status.
;
@RS_SEND        MACRO PORT
        MOV     DX,PORT
        MOV     AH,F_RS_OUT                 ;Function code
        INT     14H
        ENDM
;**********************************************************
;This macro will read the serial port status.
```

```
;The status is returned in AX and is described in the
;IBM technical reference manual on page A-23.
;
@RS_STATUS      MACRO   PORT
        MOV     DX,PORT
        MOV     AH,F_RS_STATUS  ;Function code
        INT     14H
        ENDM

;************************************************************
;The following macros are provided as an aid in programming.
;They provide many commonly used VIDEO functions.
;*******************************************************
;
;This macro provides the NEW LINE function (CR,LF) combination.
@CRLF   MACRO
        MOV     DL,0DH                  ;CARRIAGE RETURN
        @CHARDSP                        ;DISPLAY CHARACTER MACRO
        MOV     DL,0AH                  ;LINE FEED
        @CHARDSP
        ENDM
@NEW_LINE       MACRO
        @CRLF
        ENDM
;This macro provides a line feed and carriage return combination
@LFCR   MACRO
        MOV     DL,0AH                  ;LINEFEED
        @CHARDSP
        MOV     DL,0DH                  ;CARRIAGE RETURN
        @CHARDSP
        ENDM
;This macro does a carriage return only.
@CR     MACRO
        MOV     DL,0DH                  ;DO CARRIAGE RETURN
        @CHARDSP
        ENDM
;This macro does a line feed only
@LF     MACRO
        MOV     DL,0AH                  ;DO A LINE FEED ONLY
        @CHARDSP
        ENDM
```

Listing 6-2
DOSEQU.EQU

```
;Filename: DOSEQU.EQU
;MSDOS and BIOS eouate file.
;Last revised: 10-24-84
;
;(c) Gary A. Shade
;
;**************************************************************
;THIS FILE ALSO CONTAINS MANY BIOS SYSTEM CALL FUNCTION
;NUMBERS. IT SHOULD BE NOTED THAT UNLESS A "COMPATIBLE"
;PC EMULATES THESE BIOS CALLS WHICH ARE PART OF THE BIOS ROM
;IN THE IBM PC, PROGRAMS WRITTEN USING THESE CALLS MAY NOT BE
;"COMPATIBLE" WITH OTHER COMPUTERS.
;
;************ BIOS EQUATES FOR INT 10H - VIDEO FUNCTIONS ***
;
F_MODESET EQU  0          ;SETS VIDEO MODE
F_CURCHG  EQU  1          ;CHANGES CURSOR
F_CURSET  EQU  2          ;SET CURSOR POSITION
F_CURRD   EQU  3          ;GETS CURSOR POSITION
F_PENRD   EQU  4          ;READS LIGHT PEN IF INSTALLED
F_PGSET   EQU  5          ;SETS ACTIVE VIDEO PAGE
F_DIRECUP EQU  6          ;SCROLLS THE SCREEN UP
F_DIRECDWN     EQU  7     ;SCROLLS THE SCREEN DOWN
F_CHARRD  EQU  8          ;GET CHAR AT CURSOR
F_VDCHAR1 EQU  9          ;DISPLAY CHAR WITH ATTRIBUTE
F_VDCHAR2 EQU  0AH        ;DISPLAY CHARACTER W/O ATTRIBUTE
F_COLORSET     EQU  0BH   ;SET GRAPHICS COLOR
F_SET     EQU  0CH        ;SET A GRAPHICS PIXEL
F_POINT   EQU  0DH        ;GET THE COLOR OF PIXEL
F_VDCHAR3 EQU  0EH        ;OUTPUT CHARACTER AT CURSOR
F_MODERD  EQU  0FH        ;READ ACTIVE VIDEO MODE
;**********************************************************
;BIOS INT 14H function codes. Used in serial I/O
F_RS232_INIT     EQU     00H      ;Initialize serial oort
F_RS_INPUT       EQU     01H      ;Get character from oort
F_RS_OUT         EQU     02H      ;Transmit function
F_RS_STATUS      EQU     03H      ;Read the oort status
;**********************************************************
;BIOS INT 17H function codes. Used in orinter I/O.
F_PRINTER_INIT   EQU     00H      ;Initialize orinter oort
F_PRINTER_OUT_B EQU      01H      ;Send a character to the orinter
F_PRINTER_STATUS EQU     02H      ;Fetch status function

;**********************************************************
;DOS FUNCTIONS NEXT,,,,,,, INT 21H ***********************
;**********************************************************
F_TERMINATE      EQU     00       ;TERMINATE USER PROGRAM
F_WAITKEY        EQU     01H      ;WAIT FOR KEY INPUT
```

```
F_CHARDSP       EQU     02H     ;DISPLAY CHAR IN DL
F_WAITAUX       EQU     03H     ;WAIT FOR CHAR FROM AUX DEVICE
F_AUXOUT        EQU     04H     ;LIKE VALSPEAK OK? GET CHAR. FROM AUX DEVICE
F_PRINTER_OUT   EQU     05H     ;CHAR TO PRINTER
F_CON_IO        EQU     06H     ;KBD/DISPLAY ROUTINE
F_CON_INPUT     EQU     07H     ;KBD INPUT
F_CON_INPUT2    EQU     08H     ;KBD INPUT
F_VDLINE        EQU     09H     ;DISPLAY LINE TO DELIMITER
                                ;ON VIDEO DISPLAY
F_KBDLINE       EQU     0AH     ;ACCEPT A LINE OF INPUT FROM THE
                                ;KEYBOARD
F_PEEK_BUFFER   EQU     0BH     ;SEE IF CHAR IN TYPE AHEAD BUFFER
F_KBD_FLUSH     EQU     0CH     ;FLUSH TYPE AHEAD BUFFER.
F_DISK_RESET    EQU     0DH     ;FLUSH FILE BUFFERS
F_SELECT_DISK   EQU     0EH     ;LOG ON NEW DRIVE
F_OPEN          EQU     0FH     ;OPEN FCB
F_CLOSE         EQU     10H     ;CLOSE FCB
F_DIR_SEARCH    EQU     11H     ;SEARCH DIRECTORY
F_SEARCH_NEXT   EQU     12H     ;SEARCH FOR NEXT ENRTY
F_DELETE_FILES  EQU     13H     ;DELETE FILE(S)
F_READ_SRECORD  EQU     14H     ;READ SEQUENTIAL RECORD
F_WRITE_SRECORD EQU     15H     ;WRITE SEQUENTIAL RECORD
F_CREATE_FILE   EQU     16H     ;CREATE/OR OPEN FILE
F_RENAME_FILE   EQU     17H     ;RENAME AN EXISTING FILE
F_CURRENT_DISK  EQU     19H     ;READ THE CURRENT DRIVE CODE
F_SET_DTA       EQU     1AH     ;SETS THE DTA TO SPECIFIED ADDRESS
F_GET_FAT       EQU     1BH     ;GET THE FILE ALLOCATION TABLE
F_READ_RRECORD  EQU     21H     ;READ A RANDOM RECORD
F_WRITE_RRECORD EQU     22H     ;WRITE A RANDOM RECORD
F_GET_FILE_SIZE EQU     23H     ;RETURNS THE NUMBER OF RECORDS IN A FILE
F_SET_REL_RECORD EQU    24H     ;SETS RELATIVE RECORD.
F_SET_INT_VECTOR EQU    25H     ;SET'S A TYPE'S VECTOR
F_BLOCK_RREAD    EQU    27H     ;RANDOM BLOCK READ
F_BLOCK_RWRITE  EQU     28H     ;RANDOM BLOCK WRITE
F_PARSE_STRING  EQU     29H     ;PARSE STRING FOR FILENAME
F_GET_DATE      EQU     2AH     ;READ THE CURRENT DATE
F_SET_DATE      EQU     2BH     ;SET THE DATE
F_GET_TIME      EQU     2CH     ;GET THE TIME
F_SET_TIME      EQU     2DH     ;SET THE TIME
F_VERIFY_WRITE  EQU     2EH     ;SET THE VERIFY FLAG (ON/OFF)
F_GET_DTA       EQU     2FH     ;FETCH THE DTA
F_GET_VFERSION  EQU     30H     ;GET MAJOR/MINOR REV LEVEL
F_TERMINATE_RESIDENT EQU 31H    ;TERMINATE USER PROGRAM BUT
                                ;KEEP PROGRAM RESIDENT
F_SET_C_CHECK   EQU     33H     ;SET/RESET CNTL C CHECKING
F_READ_VECTOR   EQU     35H     ;READ INTERRUPT VECTOR
F_GET_FREE_SPACE EQU    36H     ;RETURNS THE FREE SPACE OF A DISK
F_MK_DIR        EQU     39H     ;CREATE A DIRECTORY
F_RM_DIR        EQU     3AH     ;REMOVE A DIRECTORY
F_CH_DIR        EQU     3BH     ;CHANGE THE CURRENT DIRECTORY
F_CREATE_FILE_2 EQU     3CH     ;CREATE A FILE (VER. 2.0)
F_OPEN_FILE_2   EQU     3DH     ;OPEN A FILE (VERSION 2.0)
F_CLOSE_FILE_2  EQU     3EH     ;CLOSE A FILE (VERSION 2.0)
F_READ_FILE_2   EQU     3FH     ;READ A NUMBER OF CHARACTERS
                                ;FROM A FILE.
F_WRITE_FILE_2  EQU     40H     ;WRITE A NUMBER OF BYTES TO
                                ;A FILE
F_DELETE_FILE_2 EQU     41H     ;DELETE THE SPECIFIED FILE
F_MOVE_POINTER  EQU     42H     ;MOVE THE FILE POINTER
F_CHANGE_ATTRIBUTES EQU 43H     ;CHANGE A FILE'S ATTRIBUTES
```

Listing 6-2 continued

```
F_DUP_HANDLE     EQU      45H      ;DUPLICATE A FILE'S HANDLE
F_DUP_HANDLE_2   EQU      46H      ;POINT EXISTING HANDLE TO
                                   ;NEW FILE
F_CURRENT_DIR    EQU      47H      ;GET THE CURRENT DIRECTORY PATH
F_ALLOCATE_MEM   EQU      48H      ;ALLOCATE MEMORY TO A PROGRAM
F_DEALLOCATE_MEM EQU      49H      ;DEALLOCATE (FREE) MEMORY
F_NEW_BLOCK_SIZE EQU      4AH      ;GROW OR SHRINK BLOCK SIZE
F_LOAD_EXECUTE   EQU      4BH      ;LOAD AND/OR EXECUTE A PROGRAM
F_EXIT           EQU      4CH      ;TERMINATE PROCESS
F_GET_RET_CODE   EQU      4DH      ;GET THE RETURN CODE FROM
                                   ;FUNCTION F_EXIT (4CH)
F_FIND_FILE      EQU      4EH      ;FIND FIRST FILE SPECIFIED
F_FIND_NEXT_FILE EQU      4FH      ;FIND THE NEXT FILE AS SPECIFIED
F_READ_VERIFY_FLAG EQU    54H      ;READ THE VERIFY FLAG
F_RENAME_FILE2   EQU      56H      ;RENAME A FILE OR PATHNAME
F_GET_SET_DT     EQU      57H      ;GET OR SET THE DTA
```

Listing 6-3
KEYDSP

```
     PAGE 60,132
     TITLE EXAMPLE KEYBOARD AND VIDEO DISPLAY PROGRAM.

        .XLIST
        INCLUDE B:DOSEQU.EQU
        .LIST

        IF1
        INCLUDE B:MACFLE.MAC
        ENDIF
;
;DISPLAY A LINE, GET KEYBOARD CHOICE.
;DISPLAY 4 LINES, GET KEYBOARD INPUT.
;DISPLAY PROMPT-WAIT FOR INPUT-DO THE FIRST PROCESS AGAIN
;USE CONTROL-C TO RETURN TO MSDOS
;
;***********************************************************
MY_STACK  SEGMENT   PARA STACK 'STACK'
               DW   100 DUP(?)
TOP_STACK      EQU  $
MY_STACK       ENDS
;***********************************************************

MY_DATA   SEGMENT   PARA 'DATA'
KBD_BUFFER     LABEL     BYTE
MAX_CHARS      DB   30
CHARS_TYPED    DB   ?               ;ACTUAL NUMBER TYPED
K_BUFF         DB   30 DUP(' ')    ;INIT TO ZERO 48 BYTES
DELIMITER      DB   '$'             ;END OF BUFFER
;
MESSAGE1  DB   'Enter what you like and I will display it.'
          DB   0AH,0DH,'$'
MESSAGE2  DB   'Hit any key to continue',0AH,0DH,'$'
MESSAGE3  DB   'This is message number 3',0AH,0DH,'$'
MESSAGE4  DB   'This is message number 4',0AH,0DH,'$'
MESSAGE5  DB   'ENTER UP TO 30 CHARS. MAX. ',0AH,0DH,'$'

MY_DATA   ENDS
;**********************************************************
MY_CODE   SEGMENT   PARA 'CODE'

START     PROC FAR
          ASSUME    CS:MY_CODE,DS:MY_DATA,SS:MY_STACK,ES:MY_DATA
          PUSH  DS                  ;SAVE RETURN
          XOR   AX,AX               ;AND OFFSET
          PUSH  AX
BEGIN:    MOV  AX,MY_CODE           ;INITIALIZE REGISTERS
          MOV  AX,MY_DATA
```

Listing 6-3 continued

```
          MOV  ES,AX
          MOV  DS,AX
;***** DEMONSTRATE MACRO USAGE. INSERT MACROS IN MAIN LINE CODE
;***** A LITTLE LIKE A HIGH LEVEL LANGUAGE.

BEGIN2:
          @VDLINE  MESSAGE1  ;DISPLAY THE FIRST MESSAGE
          @VDLINE  MESSAGE2  ;ETC.
          @WAITKEY               ;WAIT FOR KEYBOARD INPUT
          @CRLF              ;Go to the next line
          @VDLINE  MESSAGE3  ;DISPLAY MESS 3-5
          @VDLINE  MESSAGE4  ;DISPLAY MESS 4
          @VDLINE  MESSAGE5  ;MESSAGE 5
          @KBDLINE      KBD_BUFFER ;GET LINE FROM KBD
          LEA  BX,CHARS_TYPED   ;Use as an offset to last char.
                                ;entered.
          MOV  K_BUFF[BX],'$'   ;Terminate the character string.
          @VDLINE  K_BUFF       ;Display the string.
          @CRLF                 ;go to the next video line.
          CALL CLRBUF           ;Clear the keyboard buffer
          JMP   BEGIN2          ;DO IT ALL AGAIN
START   ENDP
;**************************************************************
;Clear the keyboard buffer. Intitialize to spaces.
;*****************************************************************
CLR_BUFFER      PROC    NEAR
CLRBUF:   MOV  CX,30              ;CLEAR THE KEYBOARD BUFFER
          MOV  DI,0000            ;SET DI TO ZERO OFFSET
CLR_LOOP: MOV  K_BUFF[DI],20H      ;BLANK BUFFER POSITION
          INC  DI                  ;INCREMENT POINTER
          LOOP CLR_LOOP            ;LOOP FOR 30 TIMES
          RET                      ;RETURN TO CALLER
CLR_BUFFER      ENDP               ;END OF PROCEDURE

MY_CODE   ENDS
          END  START
```

Listing 7-1
NUMBERSY.ASM

```
        PAGE    60,132
        TITLE   PROGRAM USING MACRO CALLS
        SUBTTL  Numbering Systems

        .XLIST
        INCLUDE B:DOSEQU.EQU
        .LIST

        IF1
        INCLUDE B:MACFLE.MAC
        ENDIF

;*************************************************************
;Created 08-06-84
;By: Gary A. Shade
;(c) 1984 All Rights Reserved
;
;Last Updated: 09-23-84
;
;This program will convert numbers entered from the keyboard
;to other number bases. MACFLE.ASM file,
;and the DOS equate file are used in this file.
;The numbers entered may either be decimal, binary, or hexadecimal
;numbers. You must enter all digits for each number base to be converted.
;Failure to do so will result in erroneous results.
;
;For example: to convert a decimal number, you must enter
;5 digits between 0 and 9. The maximum value you can
;enter is 65536, and the minimum is 0. Negative numbers are not
;allowed.
;
;HEXADECIMAL numbers must be a number (0-9) or a letter (A-F) only.
;The hex entry must be 4 digits in length. Example: F07F.
;
;BINARY values must be 16 digits in length and consist of either a
; '1' or a '0' (Enter only 1 or 0). No spaces between bits.
;16 bit Example: 0100011100110101
;8 bit  Example: 0000000010010101
;(You must enter 8 leading for 8 bit binary numbers)
;
;All keyboard entries must terminate with a carriage return.
;
;

MY_DATA SEGMENT PARA    'DATA'
KBD_BUFFER      LABEL   BYTE
MAX_CHARS       DB      32                  ;MAXIMUM ALLOWED
CHARS_TYPED     DB      ?                   ;NUMBER OF CHARS TYPED
```

Listing 7-1 continued

```
K_BUFF          DB      32 DUP(' ')     ;KEYBOARD BUFFER
DELIMITER       DB      '$'             ;STRING TERMINATOR

MESSAGE_0       DB      '1) Convert decimal to binary & hex. '
                DB      '$'
MESSAGE_01      DB      '2) Convert binary to decimal & hex. '
                DB      '$'
MESSAGE_02      DB      '3) Convert hexadecimal to binary and decimal. '
                DB      '$'
MESSAGE_1       DB      'Enter the decimal value you want to convert: '
                DB      '$'
MESSAGE_2       DB      'Enter the binary value you want to convert: '
                DB      '$'
MESSAGE_3       DB      'Enter the hexadecimal value you want to convert: '
                DB      '$'
MESSAGE_4       DB      'Enter '
                DB      '$'
MESSAGE_5       DB      'digits: '
                DB      '$'
MESSAGE_6       DB      'Binary '
                DB      '$'
MESSAGE_7       DB      'Decimal '
                DB      '$'
MESSAGE_8       DB      'Hexadecimal '
                DB      '$'
MESSAGE_9       DB      'Do another conversion? (Y/N) : '
                DB      '$'
HEXADECIMAL     DW      2 DUP (?)
                DB      '$'
BINARY          DW      ?
BINARY_ASCII    DB      16 DUP(20H)
                DB      '$'
DECIMAL         DB      5 DUP(?)
                DB      '$'
DECIMAL_POWERS  DW      10000,1000,100,10,1
HEXADECIMAL_POWERS DW   4096,256,16,1
MY_DATA         ENDS

;*************** DEFINE STACK AREA ********************
MY_STACK        SEGMENT PARA    STACK   'STACK'
        DW      100 DUP(?)
TOP_OF_STACK    EQU     $
MY_STACK        ENDS
;*****************************************************


;*********** MAIN PROGRAM STARTS HERE *****************
;Convert a base 16 (hex), or a base 2 (binary), or a
;base 10 (decimal) number to the other number bases.
;There are four routines which will convert value from one base
;to another.
;
;BINARY_HEXADECIMAL
;BINARY_DECIMAL
;
;DECIMAL_BINARY
;HEX_BIN_DEC
;
;*****************************************************
```

```
;Program specific equates.
ONE     EQU     31H
FOUR    EQU     34H
FIVE    EQU     35H
SIX     EQU     36H
SPACE   EQU     20H
;*******************************************************
;A macro definition which is used to display multiple messages.
;It is used here in this program and is not part of the MAC
;file on disk.

@DISP_PROMPTS   MACRO   MESS1,MESS2,MESS3,NUMBER,NUMBER_2
        @VDLINE MESS1                   ;DISPLAY PROMPT
        @LFCR                   ;CARRIAGE RETURN LINEFED
        @VDLINE MESS2                   ;DISPLAY MAX DIGIT MESSAGE
        MOV     DL,NUMBER               ;MAXIMUM CHARACTERS TO
        @CHARDSP                        ;DISPLAY IN DL
        MOV     DL,NUMBER_2             ;DISPLAY 2ND NUMBER IF ANY.
        @CHARDSP
        MOV     DL,20H                  ;SPACE CHARACTER
        @CHARDSP
        @VDLINE MESS3                   ;DISPLAY MESSAGE
        @LFCR
        @LFCR                   ;CR, LF
        ENDM

;******************************************************************
;Start of code segment
;
MY_CODE SEGMENT PARA    'CODE'
START   PROC    FAR
        ASSUME  CS:MY_CODE,DS:MY_DATA,SS:MY_STACK,ES:MY_DATA

INIT:
        PUSH    DX              ;SAVE DOS RETURN
        XOR     AX,AX           ;SAVE AN OFFSET OF ZERO
        PUSH    AX
        MOV     AX,MY_DATA      ;INITIALIZE DS AND ES
        MOV     DS,AX
        MOV     ES,AX


START_HERE:
        MOV     CX,24   ;CLEAR SCREEN USING CARRIAGE RETURNS
START_HERE1:
        @LFCR           ;CARRIAGE RETURN LINE FEED ROUTINE
        LOOP    START_HERE1     ;DO UNTIL CX= 0
MENU:
        @VDLINE MESSAGE_0       ;DISPLAY MAIN MENU
        @LFCR                   ;GO TO NEXT LINE.
        @VDLINE MESSAGE_01      ;NEXT MESSAGE
        @LFCR
        @VDLINE MESSAGE_02      ;LAST LINE OF MENU.
        @LFCR                   ;
RESPONSE:
        @WAITKEY        ;WHICH FUNCTION IS DESIRED?
        PUSH    AX              ;SAVE AX AND DO LF-CR
        @LFCR                   ;NEXT LINE
        POP     AX              ;RESTORE VALUE
        AND     AL,0FH          ;MASK BYTE
```

Listing 7-1 continued

```
        DEC     AL              ;CONVERT DEC TO OTHER?
        JZ      DEC_BIN_HEX1    ;IF SO THEN GO CONVERT
        DEC     AL              ;IF THE NUMBER WAS 2 THEN
        JZ      BIN_DEC_HEX1    ;GO THE OTHER WAY
        DEC     AL              ;IF NOT 3 THEN RESTART
        JNE     START_HERE      ;
        CALL    GET_HEX         ;OTHERWISE, DO A HEX CONVERSION
DONE_IT_2:
        CALL    FLUSH_BUFFERS   ;ZERO ALL BUFFERS
        CALL    ANOTHER_ONE             ;DO ANOTHER CONVERSIO?
        JC      MENU1                   ;IF CARRY SET THEN YES
        RET                             ;ELSE END
MENU1:
        JMP     MENU                    ;THIS IS AN ISLAND JUMP
                                        ;TO REACH THE MENU ROUTINE.
BIN_DEC_HEX1:
        CALL BIN_DEC_HEX        ;DO BINARY CONVERSIONS
        JMP     DONE_IT_2               ;COMMON RETURN POINT
DEC_BIN_HEX1:
        CALL    DEC_BIN_HEX     ;DECIMAL CONVERSIONS
        JMP     DONE_IT_2               ;ALL DONE
START   ENDP                            ;END OF MAIN PROCEDURE


;***************************************************************
;This routine will get Hexadecimal digits from the keyboard and
;convert them to decimal and binary.
;Requires procedures: HEX_BIN_DEC, BINARY_DECIMAL, AND
;                     DISP_ASCII_BINARY
;
;***************************************************************
GET_HEX PROC    NEAR
        CALL    HEX_BIN_DEC     ;GET HEX DIGITS
DSP_ASC_BIN:
        CALL    DISP_ASCII_BINARY       ;DISPLAY ASCII STRING
        @LFCR
        CALL    BINARY_DECIMAL          ;CONVERT TO DECIMAL
        @VDLINE DECIMAL                 ;DISPALAY STRING
        MOV     DL,20H                  ;SPACE
        @CHARDSP                        ;DISPLAY CHAR IN DL
        @VDLINE MESSAGE_7               ;'DECIMAL'
        RET                             ;RETURN TO CALLER
GET_HEX ENDP                            ;END OF PROCEDURE


;****** Convert decimal to binary and hexadecimal. **************
;This is the entire routine to fetch the keyboard entry of a
;decimal number (5 digits), and convert that number to binary,
;ASCII-Binary, and ASCII-Hex, and to display the conversions
;on the screen. The procedures which actually perform
;the conversions are called from this section of code.
;
DEC_BIN_HEX     PROC    NEAR
        MOV     MAX_CHARS,6     ;ACCEPT 5 CHARACTERS
        @DISP_PROMPTS   MESSAGE_1,MESSAGE_4,MESSAGE_5,FIVE,SPACE
        @KBDLINE        KBD_BUFFER      ;GET THE STRING FROM KBD.
        @LFCR                           ;DO NEW LINE
```

```
D_B_H_1:
        LEA     SI,K_BUFF                       ;MOVE INPUT FROM KEYBD BUFFER
        LEA     DI,DECIMAL                      ;TO DECIMAL STRING BUFFER
        MOV     CX,05H                          ;MOVE 5 CHARACTERS
        CLD                                     ;CLEAR DIRECTION FLAG
                                                ;AUTO INCREMENT POINTERS
        REP     MOVSB                           ;MOVE THE STRING
D_B_H_2:
        CALL    DECIMAL_BINARY                  ;CONVERT DECIMAL TO BINARY
        CALL    BINARY_HEXADECIMAL              ;CONVERT BINARY TO HEX
D_B_H_3:
        @VDLINE HEXADECIMAL                     ;DISPLAY HEXADECIMAL
        MOV     DL,20H                          ;DISPLAY A SPACE
        @CHARDSP                                ;DISPLAY CHAR. IN DL.
        @VDLINE MESSAGE_8                       ;DISPLAY 'HEX'
        @LFCR                                   ;CARRIAGE RETURN LINEFEED
        CALL    DISP_ASCII_BINARY               ;DISPLAY ASCII BINARY STRING
        RET                                     ;ALL DONE RETURN TO CALLER
DEC_BIN_HEX     ENDP                            ;END OF PROCEDURE


;***** Binary to decimal and hexadecimal routine. *****************
;Convert an ASCII-binary entry from the keyboard into ASCII-Decimal,
;and ASCII-HEX for display.
;
BIN_DEC_HEX     PROC    NEAR
        MOV     MAX_CHARS,17    ;MUST ENTER 16 CHARACTERS
        @DISP_PROMPTS   MESSAGE_2,MESSAGE_4,MESSAGE_5,ONE,SIX
        @KBDLINE        KBD_BUFFER      ;GET INPUT
        @LFCR                                   ;GO DOWN TO NEXT LINE
        LEA     SI,K_BUFF                       ;MOVE STRING TO ASCII BUFFER
        LEA     DI,BINARY_ASCII                 ;
        MOV     CX,16                           ;DO IT 16 TIMES
        CLD                                     ;CLEAR DF, AUTO INCREMENT
        REP     MOVSB                           ;MOVE BYTE STRING
        CALL ASCII_BIN_CONV                     ;CONVERT ASCII BINARY TO
                                                ;BINARY
        CALL    BINARY_HEXADECIMAL              ;CONVERT TO ASCII HEX
        @VDLINE HEXADECIMAL                     ;DISPLAY HEX STRING
        MOV     DL,20H                          ;SPACE CHARACTER
        @CHARDSP                                ;DISPLAY CHAR IN DL
        @VDLINE MESSAGE_8                       ;'HEXADECIMAL'
        @LFCR                                   ;NEW LINE
        CALL    BINARY_DECIMAL                  ;CONVERT BINARY TO DEC. STRING
        @VDLINE DECIMAL                         ;DISPLAY STRING
        MOV     DL,20H                          ;ASCII SPACE
        @CHARDSP                                ;DISPLAY IT
        @VDLINE MESSAGE_7                       ;'DECIMAL'
        @LFCR                                   ;NEWLINE
        RET
BIN_DEC_HEX     ENDP


;************** Convert hexadecimal number to decimal and binary.
;1) A 4 digit hexadecimal number is entered from the keyboard.
;2) Because the number is in ASCII, set the high order nibble of
;   each number to zero, and multiply by the base 16 power that
;   the number occupies, and accumulate the products of the 4 digits.
;   The number will have been converted to binary.
;3) To display the number, test each bit position (shift into carry),
```

Listing 7-1 continued

```
;   if it is a zero then display a 30H (ASCII zero) followed by a space.
;   If the bit value is '1', then display 31H (ASCII one) followed by a
;   space.
;4) The number which is in binary format, can then converted to
;   decimal as follows:
;  A) Divide the binary number by the highest decimal digit's power.
;  B) Use the Logical 'OR' function to convert the BCD (0-9) value
;     obtained from the division, to an ASCII digit. Store the
;     result.
;  C) Repeat the operation for the remaining powers of ten (1000, 100,
;     10, and 1).
;  D) Display the decimal string.

HEX_BIN_DEC     PROC    NEAR
        MOV     MAX_CHARS,5
        @DISP_PROMPTS   MESSAGE_3,MESSAGE_4,MESSAGE_5,FOUR,SPACE
        @KBDLINE        KBD_BUFFER      ;POINT TO KEYBOARD BUFFER
        @LFCR                           ;DO A LINEFEED, CARRIAGE RET.
        LEA     SI,K_BUFF               ;GET INPUT
        LEA     DI,HEXADECIMAL          ;AND MOVE IT
        MOV     CX,4
        CLD                             ;CLEAR DF, AUTO INCREMENT
        REP     MOVSB                   ;MOVE STRING TO BUFFER
        XOR     BX,BX                   ;CLEAR BASE REGISTER
        MOV     BINARY,BX               ;CLEAR RAM ACCUMULATOR
        XOR     SI,SI                   ;POINT TO 1ST ENTRY
        MOV     CX,4                    ;LOOP FOR 4 DIGITS
MULT_HEX:
        MOV     AL,BYTE PTR HEXADECIMAL[BX]     ;GET VALUE
        CMP     AL,41H                  ;IF >= 41H THEN IT IS
                                        ;AN ASCII A-F
        JL      ASC_NUMERAL             ;IF NOT, THEN IT'S A NUMBER
        SUB     AL,7                    ;OTHERWISE SUB 7 ADJUST FOR
                                        ;ASCII
ASC_NUMERAL:
        AND     AL,0FH          ;MASK HIGH ORDER NIBBLE
        XOR     AH,AH                   ;ZERO HIGH ORDER NIBBLE
        MUL     HEXADECIMAL_POWERS[SI]  ;MULTIPLY VALUES (UNSIGNED)
        ADD     BINARY,AX               ;IGNORE HIGH ORDER RESULT
        INC     SI                      ;POINT TO NEXT POWER
        INC     SI                      ;(WORD VALUE)
        INC     BX                      ;POINT TO NEXT HEX DIGIT
        LOOP    MULT_HEX                ;MULTIPLY ALL POSITIONS
        RET                             ;OTHERWISE QUIT.
HEX_BIN_DEC     ENDP
;zeros and then the byte value.
;
ASCII_BIN_CONV  PROC    NEAR
        XOR     AX,AX                   ;ZERO RAM LOCATION BINARY.
        MOV     BINARY,AX
        XOR     SI,SI                   ;CLEAR INDEX REGISTER
        MOV     CX,16                   ;NUMBER OF ITERATIONS.

NEXT_BYTE:
        MOV     AL,BINARY_ASCII[SI]     ;WORK ASCII MSB TO LSB
        AND     AL,0FH                  ;MASK HIGH ORDER NIBBLE
        JZ      CLEAR_CARRY             ;IF ZERO THEN CLEAR THE CARRY
        STC                             ;OTHERWISE SET THE CARRY
```

```
EXEC_LOOP:
        RCL     BINARY,1                        ;ROTATE CARRY BIT LSB TO MSB.
                                                ;LOCATION 'RAM`
        INC     SI                              ;INCREMENT POINTER
        LOOP    NEXT_BYTE                       ;DO ALL 16 BITS
        RET                                     ;RETURN WHEN ALL DIGITS DONE

CLEAR_CARRY:
        CLC                             ;CLEAR FLAG
        JMP     EXEC_LOOP                       ;CONTINUE TILL DONE
ASCII_BIN_CONV  ENDP


;***************************************************************
;This conversion routine will convert a 5 digit decimal string  *
;at RAM location 'DECIMAL' to a binary word.                    *
;The converted binary word will be stored at RAM location 'BINARY'.*
;***************************************************************
DECIMAL_BINARY  PROC    NEAR
        XOR     SI,SI                           ;ZERO SI REGISTER
        MOV     BINARY,SI                       ;ZERO RAM LOCATION
        XOR     DI,DI                           ;POINT TO TABLE
        MOV     CX,05H                          ;DO IT FOR 5 DIGITS

CONV_BIN:
        MOV     AL,DECIMAL[SI]                  ;GET 1ST DIGIT
        AND     AX,000FH                        ;MASK TO BCD
        MUL     DECIMAL_POWERS[DI]              ;MULTIPLY BY POWER
        ADD     BINARY,AX                       ;ADD TO ACCUMULATOR
        INC     SI                              ;POINT TO NEXT DECIMAL DIGIT
        INC     DI                              ;POINT TO NEXT POWER
        INC     DI
        LOOP    CONV_BIN                        ;LOOP FOR ALL DIGITS
        RET                                     ;ALL DONE
DECIMAL_BINARY  ENDP


;****************************************************************
;THIS PROCEDURE WILL CONVERT THE BINARY VALUE                   *
;IN RAM LOCATION 'BINARY' TO A 5 DIGIT DECIMAL STRING.          *
;THE ASCII DECIMAL STRING WILL BE STORED AT RAM LOCATION 'DECIMAL'. *
;****************************************************************
;CONVERT THE BINARY VALUE TO ASCII BINARY FOR
;DISPLAY.
;
DISP_ASCII_BINARY       PROC    NEAR
        MOV     AX,BINARY                       ;GET VALUE TO DISPLAY
        MOV     DI,16                           ;POINT TO LAST LOCATION
        MOV     CX,DI                           ;NUMBER OF BITS TO ROTATE

FETCH_BIN_BIT:
        ROR     AX,1                    ;ROTATE LSB n IN CARRY
        JC      A_ONE_BIT                       ;IF CARRY = 1 THEN A '1' BIT
        MOV     BINARY_ASCII[DI-1],'0'          ;STORE ASCII ZERO
        JMP     LOOPING                         ;DO NEXT LOCATION AND BIT

A_ONE_BIT:
        MOV     BINARY_ASCII[DI-1],'1'          ;SAVE AN ASCII ONE
```

Listing 7-1 continued

```
LOOPING:
        DEC     DI              ;POINT TO NEXT LOCATION
        LOOP    FETCH_BIN_BIT           ;DO FOR ALL 16 BITS

DSP_ASC_BIN_VAL:
        MOV     CX,16           ;DISPLAY 16 BYTES
        XOR     DI,DI                   ;ZERO POINTER

DSP_NEXT_CHAR:
        MOV DL,BINARY_ASCII[DI]  ;GET BYTE TO DISPLAY
        @CHARDSP                        ;DISPLAY IT.
        MOV     DL,20H                  ;DISPLAY SPACE
        @CHARDSP                        ;DISPLAY CHAR. IN DL
        INC     DI                      ;INCREMENT POINTER TO NEXT CHAR.
        LOOP    DSP_NEXT_CHAR           ;LOOP TILL ALL CHARS. DISPLAYED
        @VDLINE MESSAGE_6               ;DISPLAY WORD 'BINARY'
        @LFCR                           ;DO CARRIAGE RETURN LINE FEED.
        RET
DISP_ASCII_BINARY       ENDP


;*********** ASCII BINARY to BINARY CONVERSION *****************
;This procedure will convert an 16 bit ASCII-Binary number
;into it's true binary equivalent. The 16 byte ASCII buffer,
;BINARY_ASCII (In RAM), is assummed to hold the ASCII value as entered
;from the keyboard.
;The routine will mask the ASCII character and test B0 for a '1' or '0'.
;For examole if the ASCII byte is 31H (for a binary 1), then
;the byte is anded with 0FH, leaving a result of 01H. Thus B0 will
;contain either a '1' (if the value was 31H) or a '0'
;(if the value was 30H). The carry is then set for a '1' or cleared
;for a '0', and the carry bit rotated into the LSB of the RAM
;location BINARY. This is done for all 16 bit positions of the word.
;Data entered must be for 16 bits and not less than 16 bits. If
;it is desired to enter a binary byte value, enter 8 leading
;********************************************************************
BINARY_DECIMAL  PROC    NEAR
BIN_DEC:
        XOR     SI,SI                   ;CLEAR OTHER POINTER
        MOV     CX,05H                  ;5 BYTES IN LOOP

DEC_CLEAR:
        MOV     DECIMAL[SI],' '         ;STORE ASCII SPACE
        INC     SI                      ;NEXT LOCATION TO CLEAR
        LOOP    DEC_CLEAR               ;CLEAR 5 LOCATIONS
        MOV     CX,04H                  ;4 DIGITS IN LOOP
        XOR     SI,SI                   ;CLEAR POINTER
        MOV     DI,SI                   ;TO POWER TABLE AND STORAGE[DI]
        MOV     AX,BINARY               ;GET BINARY VALUE TO CONVERT

MAIN_CONV:
        XOR     DX,DX           ;DX MUST BE CLEAR PRIOR TO
                                        ;DIV INSTRUCTION
        DIV     DECIMAL_POWERS[SI]      ;DIVIDE AX BY TABLE ENTRY
        OR      AL,30H                  ;CONVERT TO ASCII
        MOV     DECIMAL[DI],AL          ;STORE RESULT
        INC     SI                      ;INCREMENT TABLE POINTER
        INC     SI                      ;TO NEXT DECIMAL VALUE
        INC     DI                      ;POINT TO NEXT STORAGE LOCATION
        XCHG    AX,DX                   ;GET REMAINDER FROM DX
        LOOP    MAIN_CONV               ;DO ALL 4 MAJOR DIGITS
        OR      AL,30H                  ;STORE LAST DIGIT
        MOV     DECIMAL[DI],AL          ;SAVE RESULT
        RET                             ;DONE WITH CONVERSION
BINARY_DECIMAL  ENDP

;****************************************************************
;This procedure will convert a binary value to a hexadecimal    *
;string.                                                        *
;                                                               *
;The binary value is stored in RAM at 'BINARY', and the         *
;ASCII-hex string will be stored at 'HEXADECIMAL'               *
;****************************************************************
BINARY_HEXADECIMAL      PROC    NEAR
CLR_HEX:
        MOV     CX,02           ;CLEAR FOUR BYTES
        XOR     SI,SI           ;RESET POINTER

CLR_HEX_1:
        MOV     HEXADECIMAL[SI],00H ;STORE NULL BYTE
        INC     SI                      ;POINT TO NEXT BYTE
        LOOP    CLR_HEX_1               ;CLEAR ALL LOCATIONS

 BIN_HEX_1:
;*** Convert the high order byte of the word, high order nibble
;*** followed by the low order nibble.

        XOR     SI,SI                   ;ZERO INDEX REGISTER
        MOV     AX,BINARY               ;GET VALUE TO CONVERT
        PUSH    AX                      ;SAVE IT ON STACK
        AND     AH,0F0H                 ;MASK LOW ORDER NIBBLE
        MOV     CL,04H                  ;NUMBER OF BITS TO SHIFT
        SHR     AH,CL                   ;SHIFT RIGHT 4 POSITIONS
        MOV     BYTE PTR HEXADECIMAL[SI],AH     ;SAVE MASKED VALUE
        POP     AX                      ;RETRIVE BINARY VALUE
        PUSH    AX                      ;SAVE IT AGAIN
        AND     AH,0FH                  ;MASK HIGH ORDER NIBBLE
        INC     SI
        MOV     BYTE PTR HEXADECIMAL[SI],AH     ;SAVE MASKED VALUE
;*** Now do the same for the low order byte.
BIN_HEX_2:
        POP     AX
        PUSH    AX
        AND     AL,0F0H                 ;MASK LOW ORDER NIBBLE
        MOV     CL,04H                  ;4 BITS TO SHIFT
        SHR     AL,CL                   ;SHIFT 4 BITS
        INC     SI                      ;POINT TO NEXT PLACE TO SAVE
                                        ;HEX CHARACTER
        MOV     BYTE PTR HEXADECIMAL[SI],AL     ;SAVE MASKED VALUE
        POP     AX                      ;LAST NIBBLE TO DO
        AND     AL,0FH                  ;MASK H.O. NIBBLE
        INC     SI
        MOV     BYTE PTR HEXADECIMAL[SI],AL
        MOV     CX,04                   ;NUMBER OF HEX DIGITS
        XOR     SI,SI                   ;ZERO INDEX REGISTER
NEXT_ONE:
        MOV     AL,BYTE PTR HEXADECIMAL[SI] ;GET PSEUDO HEX BYTE
        CMP     AL,0AH                  ;IS IT > OR = 0AH?
        JGE     ADD37                   ;IF IT IS ADD 37 TO BYTE
```

Listing 7-1 continued

```
        OR      BYTE PTR HEXADECIMAL[SI],30H    ;CONVERT BYTE TO ASCII
        INC     SI
        JMP     NEXT_DIGIT              ;DO TILL ALL 4 BYTES ARE
                                        ;CONVERTED.
ADD37:
        ADD     AL,37H                  ;ADD 37 TO VALUE TO
                                        ;CONVERT IT TO ASCII
        MOV     BYTE PTR HEXADECIMAL[SI],AL     ;SAVE IT
        INC     SI                      ;INCREMENT POINTER
NEXT_DIGIT:
        LOOP    NEXT_ONE        ;CONTINUE FOR ALL DIGITS
        RET
BINARY_HEXADECIMAL      ENDP


;****************************************************************
;Procedure to flush the DECIMAL, HEXADECIMAL, BINARY and
;BINARY_ASCII buffers.
;****************************************************************
FLUSH_BUFFERS   PROC    NEAR
BIN_FLUSH:
        XOR     AX,AX           ;ZERO AX
        MOV     BINARY,AX       ;BINARY STORAGE IS CLEARED

ASCII_BIN_FLUSH:
         MOV    CX,16           ;CLEAR 16 BYTES
         XOR    SI,SI                   ;ZERO POINTER
A_B_F:
        MOV     BINARY_ASCII,AH         ;STORE A ZERO
        INC     SI                      ;INCREMENT POINTER
        LOOP    A_B_F                   ;CLEAR ALL BYTES

HEX_FLUSH:
        MOV     HEXADECIMAL,AX  ;STORE ZERO WORD
        MOV     HEXADECIMAL[2],AX       ;AND AGAIN

DEC_FLUSH:
        MOV     CX,5            ;CLEAR 5 DIGITS
        XOR     SI,SI                   ;ZERO POINTER
D_F:
        MOV     DECIMAL[SI],AH          ;STORE A ZERO BYTE
        INC     SI                      ;INCREMENT POINTER
        LOOP    D_F                     ;CLEAR ALL FIVE BYTES

K_BUFF_FLUSH:
        MOV     CX,32                   ;CLEAR 32 BYTES
        MOV     AH,20H                  ;ASCII SPACE
        XOR     SI,SI                   ;CLEAR POINTER
K_B_F:
        MOV     K_BUFF,AH               ;STORE BYTE
        INC     SI                      ;INCREMENT POINTER
        LOOP    K_B_F                   ;CLEAR ALL BYTES IN BUFFER
        RET
FLUSH_BUFFERS   ENDP


;*****************************************************************
;Procedure to ask user if they want another conversion to take place.
;On return, the carry flag will be set if the user wants to do another
```

```
;conversion. If 'NO', then the carry is reset.
;*****************************************************************
ANOTHER_ONE     PROC    NEAR
        @LFCR                   ;NEW LINE FUNCTION
        @VDLINE MESSAGE_9       ;DISPLAY PROMPT
        @WAITKEY                ;WAIT FOR INPUT
        CMP     AL,'Y'          ;IS IT YES?
        JZ      SET_CF          ;IF IT IS THEN SET CARRY
        CMP     AL,'y'
        JZ      SET_CF

;ANY OTHER KEY ENTRY WILL CAUSE THE PROGRAM TO END.

        @LFCR                   ;NEW LINE FUNCTION
        CLC                     ;CLEAR THE CARRY FLAG
        RET                     ;AND RETURN
SET_CF:
        @LFCR                   ;DO NEW LINE
        STC                     ;SET CARRY FLAG
        RET
ANOTHER_ONE     ENDP

MY_CODE ENDS
        END     START
```

Listing 8-1
NEW_TYPE.ASM

```
                         PAGE 60,132
                         TITLE NEW_TYPE.ASM
                         SUBTTL SEQUENTIAL READ EXAMPLE

                         .XLIST
                         INCLUDE B:DOSEQU.EQU
                         .LIST

                         IF1
                         INCLUDE B:MACFLE.MAC
                         ENDIF

;**************************************************************
;Sequential Read Example.
;By: Gary A. Shade
;Date: 07-10-84
;Last Updated: 08-21-84
;(c) All rights reserved.
;This program will sequentially read a text file from disk.
;The program is similar to the TYPE command available at the
;user level to MSDOS. The program sets up the DTA and prompts
;the user for a drive and filename spec. This program will
;work with either MSDOS 1.x or 2.0.
;
;The program terminates when an end of file character is encountered
;(HEX 1AH).
;
;
;
;****************************************************************
;Define the data segement.
;
MY_DATA SEGMENT PARA     'DATA'
KEYBOARD        LABEL    BYTE
MAX_CHARS       DB       ?                ;Maximum characters
                                          ;for keyboard line input
                                          ;function.
CHARS_TYPED     DB       ?                ;Actual number of characters
                                          ;typed by user.
K_BUFF          DB       20 DUP(20H)      ;20 byte buffer

DISK_BUFFER     DB       512 DUP(0)       ;Disk transfer area.
                DB       '$'              ;Terminate for display function
FCB_1           LABEL    BYTE             ;Disk FCB definition
DRIVE_NUMBER    DB       0                ;0 = default drive
                                          ;1 = A
                                          ;2 = B
FILE_NAME       DB       8 DUP(0)         ;Filename spec.
```

```
EXTENSION       DB      3 DUP(0)        ;Extension for filename
                                        ;(i.e. .ASM)
BLOCK_NUMBER    DW      0               ;Current block number
RECORD_SIZE     DW      0               ;Logical record size
FILE_SIZE       DW      2 DUP(0)        ;Double word for size of
                                        ;file in bytes.
FILE_DATE       DW      0               ;Date of last access.
RESERVED        DT      0               ;Ten reserved bytes for DOS
RECORD_NUMBER   DB      0               ;Current record number
RANDOM_NUMBER   DW      2 DUP(0)        ;4 byte random record number

MESSAGE1        DB      'Enter the drive and filename '
                DB      '$'
MESSAGE2        DB      'Invalid filename, please reenter. '
                DB      '$'
MESSAGE3        DB      'ERROR! Operation aborted.'
                DB      '$'
MESSAGE4        DB      'Sequential read complete.'
                DB      '$'
MESSAGE5        DB      'File not found.'
                DB      '$'

MY_DATA ENDS
;********************************************************************
;Define stack area.
;
MY_STACK        SEGMENT PARA    STACK   'STACK'
        DW      32 DUP(0)       ;64 words of stack.
MY_STACK        ENDS
;****************************************************************
;Define the program.
;
MY_CODE SEGMENT PARA    'CODE'
START   PROC    FAR
        ASSUME  CS:MY_CODE,DS:MY_DATA,ES:MY_DATA,SS:MY_STACK

START_1:
        PUSH    DS              ;Save segment of PSP
        XOR     AX,AX           ;Zero offset to stack
        PUSH    AX
        MOV     AX,MY_DATA      ;Initialize DS and ES
        MOV     DS,AX
        MOV     ES,AX

START_2:
        @SCROLL 06,00,18H,4FH,00,00,07  ;Clear the screen
        @VDLINE MESSAGE1                ;Ask for the filename
        @CRLF                           ;Go to the next line.
        MOV     MAX_CHARS,15            ;15 Characters maximum
                                        ;as in B:TESTFILE.TXT<CR>
GET_NAME:
        @KBDLINE        KEYBOARD        ;Get line from keyboard.

        @PARSE_STRING K_BUFF,FCB_1,03H  ;Parse the filename
                                        ;and place it in the FCB
        CALL    OPEN_FILE               ;Go open the file
        JC      START_2                 ;If the carry is set then
                                        ;TRY AGAIN! File not found.
SET_UP:
        @SET_DTA DISK_BUFFER            ;Set the Disk transfer address
```

Listing 8-1 continued

```
        MOV     RECORD_NUMBER,0         ;1st record
        MOV     BLOCK_NUMBER,0          ;1st block
        MOV     RECORD_SIZE,512         ;Record size equal to 512 bytes
READ_RECORD:
        @READ_SRECORD FCB_1             ;Sequential read function
                                        ;using the FCB defined
                                        ;in the data segment.
        CMP     AL,00H                  ;If zero then OK.
        JZ      CONTINUE
READ_ERROR:
        DEC     AL                      ;If AL = 1 then EOF encountered
                                        ;no data in record.
        JZ      EOF_FOUND
        DEC     AL                      ;If AL = 2 then no room in the
                                        ;DTA
        JZ      DISK_ERROR
        DEC     AL                      ;If AL = 3 then EOF encountered
                                        ;partial record read.
        JNZ     DISK_ERROR              ;If not 03 then unrecoverable
                                        ;disk error occurred.
        XOR     DI,DI                   ;Zero the index register
DSP_PARTIAL:
        MOV     DL,DISK_BUFFER[DI]      ;Get the character from buffer
        CMP     DL,1AH                  ;EOF character?
        JZ      EOF_FOUND               ;exit if so.
        @CHARDSP                        ;Display the character.
        INC     DI                      ;Point to the next character.
        JMP     SHORT   DSP_PARTIAL     ;Continue.
CONTINUE:
        @VDLINE DISK_BUFFER             ;Display the record.
        @CRLF
        JMP     READ_RECORD             ;Go read the next record.
EOF_FOUND:
        @VDLINE MESSAGE4                ;Display end of program message.
        @CRLF
        JMP     SHORT   EXIT            ;DONE
DISK_ERROR:
        @VDLINE MESSAGE3                ;Display error message
        @CRLF
EXIT:
        @CLOSE FCB_1                    ;Close the file.
        RET
START   ENDP                            ;End of this procedure

;***********************************************************
;Procedure to open a file specified in the FCB.
;
OPEN_FILE       PROC    NEAR
        @OPEN   FCB_1           ;Open the file
        CMP     AL,00H          ;If zero then open was successful
        JZ      RESET_CARRY     ;Reset the carry to indicate to
                                ;calling procedure the operation
                                ;worked.
        @VDLINE MESSAGE2        ;Reenter filename.
        @CRLF
        XOR     DI,DI           ;Clear the keyboard buffer.
        MOV     CX,15           ;Clear the keyboard buffer
```

```
CLR:
        MOV     K_BUFF[DI],20H   ;
        INC     DI               ;Point to the next byte.
        LOOP    CLR              ;Continue till CX = 0
        @CLOSE  FCB_1            ;Close the file
        STC                      ;Set the carry to indicate error
        JMP     SHORT   DONE_O_F ;and return
RESET_CARRY:
        CLC                      ;Clear the carry
DONE_O_F:
        RET                      ;And return.
OPEN_FILE       ENDP
;*************************************************************
MY_CODE ENDS
        END     START
```

Listing 8-2
NEW_COPY.ASM

```
                        PAGE 60,132
                        TITLE NEW_COPY.ASM
                        SUBTTL SEQUENTIAL READ/WRITE EXAMPLE

                        .XLIST
                        INCLUDE B:DOSEQU.EQU
                        .LIST

                        IF1
                        INCLUDE B:MACFLE.MAC
                        ENDIF

;************************************************************
;Sequential Read/Write Example.
;By: Gary A. Shade
;Date: 07-10-84
;Last Updated: 08-21-84
;(c) All rights reserved.
;
;
;This program will copy the contents of one file to another.
;The user is prompted for the source filename, and then for the
;destination filename.
;If the destination file already exists, the file length is set
;to zero, effectively erasing the previous contents of the file.
;If the source filename is not found, the program will terminate.
;
;This function is similar to the copy command used in MSDOS.
;It serves to illustrate the sequential read and write functions
;available under MSDOS.
;
;The macros used in this file can be found in the MACFLE.MAC
;file. MACFLE.MAC expects that the DOSEQU.EQU file is present
;in the assembly.
;
;**************************************************************
;Define the data segement.
;
MY_DATA SEGMENT PARA    'DATA'
KEYBOARD        LABEL   BYTE
MAX_CHARS       DB      ?               ;Maximum characters
                                        ;for keyboard line inout
                                        ;function.
CHARS_TYPED     DB      ?               ;Actual number of characters
                                        ;typed by user.
K_BUFF          DB      32 DUP(?)       ;32 byte buffer

DISK_BUFFER     DB      0               ;Source and destination
```

```
                                         ;disk transfer area.

FCB_1              LABEL    BYTE         ;Disk FCB definition
DRIVE_NUMBER       DB       0            ;0 = default drive
                                         ;1 = A
                                         ;2 = B
FILE_NAME          DB       8 DUP(0)     ;Filename spec.
EXTENSION          DB       3 DUP(0)     ;Extension for filename
                                         ;(i.e. .ASM)
BLOCK_NUMBER       DW       0            ;Current block number
RECORD_SIZE        DW       0            ;Logical record size
FILE_SIZE          DW       2 DUP(0)     ;Double word for size of
                                         ;file in bytes.
FILE_DATE          DW       0            ;Date of last access.
RESERVED           DT       0            ;Ten reserved bytes for DOS
RECORD_NUMBER      DB       0            ;Current record number
RANDOM_NUMBER      DW       2 DUP(0)     ;4 byte random record number

FCB_2    LABEL     BYTE                  ;Destination FCB
DRIVE_NUMBER_D     DB       0            ;0 = default drive
                                         ;1 = A
                                         ;2 = B
FILE_NAME_D        DB       8 DUP(0)     ;Filename spec.
EXTENSION_D        DB       3 DUP(0)     ;Extension for filename
                                         ;(i.e. .ASM)
BLOCK_NUMBER_D     DW       0            ;Current block number
RECORD_SIZE_D      DW       0            ;Logical record size
FILE_SIZE_D        DW       2 DUP(0)     ;Double word for size of
                                         ;file in bytes.
FILE_DATE_D        DW       0            ;Date of last access.
RESERVED_D         DT       0            ;Ten reserved bytes for DOS
RECORD_NUMBER_D    DB       0            ;Current record number
RANDOM_NUMBER_D    DW       2 DUP(0)     ;4 byte random record number


MESSAGE1           DB       'Enter the source drive and filename '
                   DB       '$'
MESSAGE1_1         DB       'Enter the destination drive and filename '
                   DB       '$'
MESSAGE2           DB       'Invalid filename, please reenter. '
                   DB       '$'
MESSAGE3           DB       'ERROR! Operation aborted.'
                   DB       '$'
MESSAGE4           DB       'File copy complete.'
                   DB       '$'
MESSAGE5           DB       'File not found.'
                   DB       '$'
MESSAGE6           DB       'New File!'
                   DB       '$'
MESSAGE7           DB       'Existing File Over Written. '
                   DB       '$'
MESSAGE8           DB       'Copy has started .. '
                   DB       '$'
MESSAGE9           DB       'Disk full, operation aborted!'
                   DB       '$'
MESSAGE10          DB       'Not enough room in DTA to write 1 record. '
                   DB       'Operation aborted.'
                   DB       '$'
BYTE_COUNT         DB       0    ;Used as a byte counter during disk writes.
MY_DATA ENDS
```

Listing 8-2 continued

```
;*******************************************************************
;Define stack area.
;
MY_STACK        SEGMENT PARA    STACK   'STACK'
        DW      32 DUP(0)       ;64 words of stack.
MY_STACK        ENDS
;***************************************************************
;Define the program.
;
MY_CODE SEGMENT PARA    'CODE'
START   PROC    FAR
        ASSUME  CS:MY_CODE,DS:MY_DATA,ES:MY_DATA,SS:MY_STACK

START_1:
        PUSH    DS              ;Save segment of PSP
        XOR     AX,AX           ;Zero offset to stack
        PUSH    AX
        MOV     AX,MY_DATA      ;Initialize DS and ES
        MOV     DS,AX
        MOV     ES,AX

START_2:
        @SCROLL 06,00,18H,4FH,00,00,07  ;Clear the screen
        @CURSET 0,0,0                   ;Set the cursor.
        @VDLINE MESSAGE1                ;Ask for the source filename
        @CRLF                           ;Go to the next line.
        MOV     MAX_CHARS,15            ;15 Characters maximum
                                        ;as in B:TESTFILE.TXT(CR)
GET_NAME:
        @KBDLINE        KEYBOARD        ;Get line from keyboard.

        @CRLF
        @PARSE_STRING K_BUFF,FCB_1,03H  ;Parse the filename
                                        ;and place it in the FCB
        CALL    OPEN_FILE               ;Go open the file
        JNC     SET_UP                  ;If the carry is set then
                                        ;TRY AGAIN! File not found.
        @VDLINE MESSAGE4                ;Display error message
        @CRLF
        JMP     SHORT START_2           ;Try again.
SET_UP:
        @SET_DTA DISK_BUFFER            ;Set the Disk transfer address
                                        ;for the destination drive
        MOV     RECORD_NUMBER,0         ;1st record
        MOV     BLOCK_NUMBER,0          ;1st block
        MOV     RECORD_SIZE,1           ;Record size equal to 1 byte
GET_DESTINATION:
        CALL    GET_SECOND              ;Get the second drive spec.
        @VDLINE MESSAGE8                ;Display copy started message
        @CRLF
READ_RECORD:
        @READ_SRECORD FCB_1             ;Sequential read function
                                        ;using the FCB defined
                                        ;in the data segment.
        CMP     AL,00H                  ;If zero then OK.
        JZ      CONTINUE
READ_ERROR:
        CMP     AL,01H                  ;If AL = 1 then EOF encountered
                                        ;no data in record.
        JE      EOF_FOUND
ERR_3:
        JMP     DISK_ERROR                      ;disk error occurred.
                                                ;Else a partial record was
                                                ;read.
CONTINUE:
        @WRITE_SRECORD FCB_2                    ;Write the record to disk
        CMP     AL,00H                          ;00 = Success
        JNZ     WRITE_ERROR                     ;If not zero then there was an error
        INC     BYTE_COUNT                      ;Increment memory.
        JNZ     GET_NEXT                        ;If byte rolls over to zero
                                                ;then display asterisk
                                                ;represents 256 bytes written
                                                ;so far.
        MOV     DL,'*'                          ;Display * for snow
        @CHARDSP
GET_NEXT:
        JMP     READ_RECORD                     ;Go read the next record.
EOF_FOUND:
        MOV     DL,'-'                          ;Show a - for end of file
        @CHARDSP
        @CRLF                                   ;next display line
        @VDLINE MESSAGE4                        ;Display end of program message.
        @CRLF
        JMP     SHORT   EXIT                    ;DONE
WRITE_ERROR:
        DEC     AL                              ;If AL = 01 then disk full
        JNZ     CHK_DTA_ERROR
        @VDLINE MESSAGE9                        ;Display error message
        @CRLF
        JMP     SHORT EXIT                      ;Leave program
CHK_DTA_ERROR:

        DEC     AL                              ;If AL = 02 then not enough
                                                ;room in the DTA
        JNZ     DISK_ERROR                      ;Undefined condition if AL>2
        @VDLINE MESSAGE10                       ;Display error message
        @CRLF
        JMP     SHORT   EXIT
DISK_ERROR:
        @VDLINE MESSAGE3                        ;Display error message
        @CRLF
EXIT:

        @CLOSE FCB_1                            ;Close the file.
        @CLOSE FCB_2                            ;Close the file.
        RET
START   ENDP                                    ;End of this procedure

;************************************************************
;Procedure to open a file specified in the FCB.
;
OPEN_FILE       PROC    NEAR
        @OPEN   FCB_1           ;Open the file
        CMP     AL,00H          ;If zero then open was successful
        JZ      RESET_CARRY     ;Reset the carry to indicate to
                                ;calling procedure the operation
                                ;worked.
        @VDLINE MESSAGE2        ;Reenter filename.
        @CRLF
        CALL    CLR_KEYS        ;Clear the keyboard buffer
```

Listing 8-2 continued

```
        @CLOSE  FCB_1                   ;Close the file
        STC                             ;Set the carry to indicate error
        JMP     SHORT   DONE_O_F ;and return
RESET_CARRY:
        CLC                             ;Clear the carry
DONE_O_F:
        RET                             ;And return.
OPEN_FILE       ENDP
;*************************************************************
;Procedure to obtain destination filename and create the file.
;
GET_SECOND      PROC    NEAR

        @VDLINE MESSAGE1_1                      ;Ask for the destination filename
        @CRLF                                   ;Go to the next line.
        MOV     MAX_CHARS,15                    ;15 Characters maximum
                                                ;as in B:TESTFILE.TXT<CR>
GET_NAME_D:
        CALL    CLR_KEYS                        ;Clear keyboard buffer
        @KBDLINE        KEYBOARD                ;Get line from keyboard.

        @CRLF
        @PARSE_STRING K_BUFF,FCB_2,03H  ;Parse the filename
                                                ;and place it in the FCB
        @CREATE_FILE    FCB_2                   ;Open the file
        CMP     AL,00H                  ;If zero then open existing file
        JNZ     NEW_FILE                ;If not zero then a new file
        @VDLINE MESSAGE7                ;Existing File message
        @CRLF
SET_UP_D:
        MOV     RECORD_NUMBER_D,0               ;1st record
        MOV     BLOCK_NUMBER_D,0                ;1st block
        MOV     RECORD_SIZE_D,1                 ;Record size equal to 1 byte
        JMP     SHORT   DONE_O_F_D              ;Return with carry clear
NEW_FILE:
        @VDLINE MESSAGE6                ;New file
        @CRLF
        JMP     SHORT   SET_UP_D ;Set up destination FCB
DONE_O_F_D:
        RET                             ;And return.

GET_SECOND      ENDP

;***********************************************************
;Clear the keyboard buffer area
CLR_KEYS        PROC    NEAR
        XOR     DI,DI                   ;Clear the keyboard buffer.
        MOV     CX,15                   ;Clear the keyboard buffer
CLR:
        MOV     K_BUFF[DI],20H          ;
        INC     DI                      ;Point to the next byte.
        LOOP    CLR                     ;Continue till CX = 0
        RET                             ;All done
CLR_KEYS        ENDP
;***********************************************************

MY_CODE ENDS
        END     START
```

Listing 8-3
FAST_CPY.ASM

```
                        PAGE 60,132
                        TITLE FAST_COP.ASM
                        SUBTTL RANDOM BLOCK READ/WRITE EXAMPLE

                        .XLIST
                        INCLUDE B:DOSEQU.EQU
                        .LIST

                        IF1
                        INCLUDE B:MACFLE.MAC
                        ENDIF

;************************************************************
;Random Block Read/Write Examole.
;By: Gary A. Shade
;Date: 07-10-84
;Last Updated: 08-21-84
;(c) All rights reserved.
;
;
;This program will copy the contents of one file to another.
;The user is prompted for the source filename, and then for the
;destination filename.
;If the destination file already exists, the file length is set
;to zero, effectively erasing the previous contents of the file.
;If the source filename is not found, the program will terminate.
;
;This function is similar to the copy command used in MSDOS.
;It serves to illustrate the sequential read and write functions
;available under MSDOS.
;
;The macros used in this file can be found in the MACFLE.MAC
;file. MACFLE.MAC expects that the DOSEQU.EQU file is oresent
;in the assembly.
;
;*************************************************************
;Define the data segement.
;
MY_DATA SEGMENT PARA    'DATA'
KEYBOARD        LABEL   BYTE
MAX_CHARS       DB      ?                ;Maximum characters
                                         ;for keyboard line inout
                                         ;function.
CHARS_TYPED     DB      ?                ;Actual number of characters
                                         ;typed by user.
K_BUFF          DB      32 DUP(?)        ;32 byte buffer

DISK_BUFFER     DB      7FFFH DUP(0)     ;Source and destination
```

Listing 8-3 continued

```
                                               ;disk transfer area.

FCB_1           LABEL   BYTE                   ;Disk FCB definition
DRIVE_NUMBER    DB      0                      ;0 = default drive
                                               ;1 = A
                                               ;2 = B
FILE_NAME       DB      8 DUP(0)               ;Filename spec.
                                               ;1 = A
                                               ;2 = B
FILE_NAME       DB      8 DUP(0)               ;Filename spec.
EXTENSION       DB      3 DUP(0)               ;Extension for filename
                                               ;(i.e. .ASM)
BLOCK_NUMBER    DW      0                      ;Current block number
RECORD_SIZE     DW      0                      ;Logical record size
FILE_SIZE       DW      2 DUP(0)               ;Double word for size of
                                               ;file in bytes.
FILE_DATE       DW      0                      ;Date of last access.
RESERVED        DT      0                      ;Ten reserved bytes for DOS
RECORD_NUMBER   DB      0                      ;Current record number
RANDOM_NUMBER   DW      2 DUP(0)               ;4 byte random record number

FCB_2    LABEL  BYTE                           ;Destination FCB
DRIVE_NUMBER_D  DB      0                      ;0 = default drive
                                               ;1 = A
                                               ;2 = B
FILE_NAME_D     DB      8 DUP(0)               ;Filename spec.
EXTENSION_D     DB      3 DUP(0)               ;Extension for filename
                                               ;(i.e. .ASM)
BLOCK_NUMBER_D  DW      0                      ;Current block number
RECORD_SIZE_D   DW      0                      ;Logical record size
FILE_SIZE_D     DW      2 DUP(0)               ;Double word for size of
                                               ;file in bytes.
FILE_DATE_D     DW      0                      ;Date of last access.
RESERVED_D      DT      0                      ;Ten reserved bytes for DOS
RECORD_NUMBER_D DB      0                      ;Current record number
RANDOM_NUMBER_D DW      2 DUP(0)               ;4 byte random record number


MESSAGE1        DB      'Enter the source drive and filename '
                DB      '$'
MESSAGE1_1      DB      'Enter the destination drive and filename '
                DB      '$'
MESSAGE2        DB      'Invalid filename, please reenter. '
                DB      '$'
MESSAGE3        DB      'ERROR! Operation aborted.'
                DB      '$'
MESSAGE4        DB      'File copy complete.'
                DB      '$'
MESSAGE5        DB      'File not found.'
                DB      '$'
MESSAGE6        DB      'New File!'
                DB      '$'
MESSAGE7        DB      'Existing File Over Written. '
                DB      '$'
MESSAGE8        DB      'Copy has started .. '
                DB      '$'
MESSAGE9        DB      'Disk full, operation aborted!'
                DB      '$'
MY_DATA ENDS
```

```
;*******************************************************************
;Define stack area.
;
MY_STACK        SEGMENT PARA    STACK   'STACK'
        DW      32 DUP(0)       ;64 words of stack.
MY_STACK        ENDS
;***************************************************************
;Define the program.
;
MY_CODE SEGMENT PARA    'CODE'
START   PROC    FAR
        ASSUME  CS:MY_CODE,DS:MY_DATA,ES:MY_DATA,SS:MY_STACK

START_1:
        PUSH    DS              ;Save segment of PSP
        XOR     AX,AX           ;Zero offset to stack
        PUSH    AX
        MOV     AX,MY_DATA      ;Initialize DS and ES
        MOV     DS,AX
        MOV     ES,AX

START_2:
        @SCROLL 06,00,18H,4FH,00,00,07  ;Clear the screen
        @CURSET 0,0,0                   ;Set the cursor.
        @VDLINE MESSAGE1                ;Ask for the source filename
        @CRLF                           ;Go to the next line.
        MOV     MAX_CHARS,15            ;15 Characters maximum
                                        ;as in B:TESTFILE.TXT<CR>
GET_NAME:
        @KBDLINE        KEYBOARD        ;Get line from keyboard.

        @CRLF
        @PARSE_STRING K_BUFF,FCB_1,03H  ;Parse the filename
                                        ;and place it in the FCB
        CALL    OPEN_FILE               ;Go open the file
        JNC     SET_UP                  ;If the carry is set then
                                        ;TRY AGAIN! File not found.
        @VDLINE MESSAGE4                ;Display error message
        @CRLF
        JMP     SHORT START_2           ;Try again.
SET_UP:
        @SET_DTA DISK_BUFFER            ;Set the Disk transfer address
                                        ;for the destination drive
        MOV     RECORD_NUMBER,0         ;1st record
        MOV     BLOCK_NUMBER,0          ;1st block
        @SET_REL_RECORD FCB_1           ;Set the random relative record
                                        ;field in the FCB.
GET_DESTINATION:
        CALL    GET_SECOND              ;Get the second drive spec.
        @VDLINE MESSAGE8                ;Display copy started message
        @CRLF
READ_RECORD:

        @BLOCK_RREAD FCB_1,7FFFH,1      ;Random Block Read
                                        ;using the FCB defined
                                        ;in the data segment.
        CMP     AL,00H                  ;If zero then OK.
        JZ      CONTINUE
```

Listing 8-3 continued

```
READ_ERROR:
        CMP     AL,01H                  ;If AL = 1 then EOF encountered
                                        ;LAST RECORD COMPLETE
        JE      PARTIAL_3
ERR_3:
        CMP     AL,03H                  ;If AL = 02H then EOF with
                                        ;partial record read.
        JZ      PARTIAL_3
        JMP     DISK_ERROR              ;disk error occurred.
PARTIAL_3:
        CMP     CX,0                    ;If no records read then skip
        JZ      EOF_FOUND
        MOV     BX,CX                   ;Put record count in BX
        @BLOCK_RWRITE FCB_2,BX,1        ;Write the partial block to disk
        JMP     SHORT EOF_FOUND
CONTINUE:
        @BLOCK_RWRITE FCB_2,7FFFH,1     ;Write the record to disk
        CMP     AL,00H                  ;00 = Success
        JNZ     WRITE_ERROR             ;If not zero then there was an error
GET_NEXT:
        JMP     READ_RECORD             ;Go read the next record.
EOF_FOUND:
        MOV     DL,'-'                  ;Show a - for end of file
        @CHARDSP
        @CRLF                           ;next display line
        @VDLINE MESSAGE4                ;Display end of program message.
        @CRLF
        JMP     SHORT   EXIT            ;DONE
WRITE_ERROR:
        DEC     AL                      ;If AL = 01 then disk full
        JNZ     CHK_SIZE_ERROR          ;Disk full error
        @VDLINE MESSAGE9                ;Display error message
        @CRLF
        JMP     SHORT EXIT              ;Leave program
CHK_SIZE_ERROR:

        @VDLINE MESSAGE9                ;Display error message
        @CRLF
        JMP     SHORT   EXIT
DISK_ERROR:
        @VDLINE MESSAGE3                ;Display error message
        @CRLF
EXIT:

        @CLOSE FCB_1                    ;Close the file.
        @CLOSE FCB_2                    ;Close the file.
        RET
START   ENDP                            ;End of this procedure

;************************************************************
;Procedure to open a file specified in the FCB.
;
OPEN_FILE       PROC    NEAR
        @OPEN   FCB_1           ;Open the file
        CMP     AL,00H          ;If zero then open was successful
        JZ      RESET_CARRY     ;Reset the carry to indicate to
                                ;calling procedure the operation
                                ;worked.
        @VDLINE MESSAGE2        ;Reenter filename.
        @CRLF
        CALL    CLR_KEYS            ;Clear the keyboard buffer
        @CLOSE  FCB_1               ;Close the file
        STC                         ;Set the carry to indicate error
        JMP     SHORT   DONE_O_F ;and return
RESET_CARRY:
        CLC                         ;Clear the carry
DONE_O_F:
        RET                         ;And return.
OPEN_FILE       ENDP
;***********************************************************
;Procedure to obtain destination filename and create the file.
;
GET_SECOND      PROC    NEAR

        @VDLINE MESSAGE1_1                  ;Ask for the destination filename
        @CRLF                               ;Go to the next line.
        MOV     MAX_CHARS,15                ;15 Characters maximum
                                            ;as in B:TESTFILE.TXT<CR>
GET_NAME_D:
        CALL    CLR_KEYS                    ;Clear keyboard buffer
        @KBDLINE        KEYBOARD            ;Get line from keyboard.

        @CRLF
        @PARSE_STRING K_BUFF,FCB_2,03H      ;Parse the filename
                                            ;and place it in the FCB
        @CREATE_FILE    FCB_2               ;Open the file
        CMP     AL,00H              ;If zero then open existing file
        JNZ     NEW_FILE            ;If not zero then a new file
        @VDLINE MESSAGE7            ;Existing File message
        @CRLF
SET_UP_D:
        MOV     RECORD_NUMBER_D,0           ;1st record
        MOV     BLOCK_NUMBER_D,0            ;1st block
        MOV     RECORD_SIZE_D,1             ;Record size equal to 1 byte
        @SET_REL_RECORD FCB_2               ;Set the relative record field
        JMP     SHORT   DONE_O_F_D          ;Return with carry clear
NEW_FILE:
        @VDLINE MESSAGE6            ;New file
        @CRLF
        JMP     SHORT   SET_UP_D ;Set up destination FCB
DONE_O_F_D:
        RET                         ;And return.

GET_SECOND      ENDP

;***********************************************************
;Clear the keyboard buffer area
CLR_KEYS        PROC    NEAR

        XOR     DI,DI               ;Clear the keyboard buffer.
        MOV     CX,15               ;Clear the keyboard buffer
CLR:
        MOV     K_BUFF[DI],20H      ;
        INC     DI                  ;Point to the next byte.
        LOOP    CLR                 ;Continue till CX = 0
        RET                         ;All done
CLR_KEYS        ENDP
;***********************************************************

MY_CODE ENDS
        END     START
```

Listing 8-4
DIRREAD.ASM

```
        PAGE    60,132
        TITLE   Directory Read Routine.
        SUBTTL Directory Sort.

        .XLIST
        INCLUDE B:DOSEQU.EQU
        .LIST

        IF1
        INCLUDE B:MACFLE.MAC
        ENDIF

;This program will read and sort the directory from the
;specified disk drive. The number of entries which are
;displayed is limited to the the screen size.
;Entries are listed four accross, for a total of
; 4 x 24 directory entries.
;This program uses DOS interrupt type 25H for
;absolute disk reads.
;DIR_SORT is the routine which will alphabetize and sort
;the entries prior to display.
;*****************************************************
; Possible disk formats are:
;Single Sided/8 sectors per track (SS8ST)
;Double Sided/8 sectors per track (DS8ST)
;Single Sided/9 sectors per track (SS9ST)
;Double Sided/9 sectors per track (DS9ST)
;
;Use the first byte of FAT to determine the drive type:
;FF = DS8ST
;FE = SS8ST
;FD = DS9ST
;FC = SS9ST
;F8 = Hard disk.
;
;*******************************************************
;The procedure GET_DRIVE_TYPE will set the parameters for
;the drive type requested dependent upon the 1st byte of the
;FAT.
;Version 1.0 formats.
;***************************************************
;       IF      SS8ST
;DIR_SEC = 4            ;Number of directory sectors
;DIR_BEG = 3            ;Where the directory begins
;ENTRIES = 64           ;Number of directory entries
;       ELSE
;       IF      DS8ST
;DIR_SEC = 7
```

```
;DIR_BEG = 3
;ENTRIES = 112
;*****************************************************
;Version 2.0 and 2.1
;*****************************************************
;       IF      SS9ST
;DIR_SEC = 4
;DIR_BEG = 5
;ENTRIES = 64
;       ELSE
;       IFE     DS9ST
;DIR_SEC = 7
;DIR_BEG = 5
;ENTRIES = 112
;****************************************************
;Define Stack Segment.
MY_STACK        SEGMENT PARA    STACK   'STACK'
        DW      100 DUP(?)
TOP_OF_STACK EQU        $
MY_STACK        ENDS
;*******************************************************
MY_DATA SEGMENT PARA    'DATA'

;***** Keyboard buffer ********************************
K_BUFFER        LABEL   BYTE
MAX_CHARS       DB      0       ;INITIALIZE TO 0.
CHARS_TYPED     DB      0       ;SAME HERE.
K_BUFF          DB      32 DUP(' ')     ;32 BYTES = ASCII SPACE

;********* Buffer for Directory read ********************
FCB             DB      40 DUP(0)       ;File Control Block
DIR_BUFF        DB      4096 DUP(0)     ;Disk Transfer Address
DIR_BUF2        DB      2048 DUP(0)     ;Secondary buffer used
                                        ;for sorting.
MESSAGE1        DB      'Disk Error, returning to MSDOS'
                DB      '$'             ;MESSAGE TERMINATOR.
MESSAGE2        DB      'Which drive (A, B)? '
                DB      '$'
MESSAGE3        DB      'Invalid entry! Please reenter '
                DB      '$'
ENTRY_COUNTER   DB      00H             ;Entry count.
ERR_COUNT       DB      00H             ;Error counter
DRIVE           DB      00H             ;Drive number (0=A)
MIN             DB      00H             ;Minimum value for bounds
MAX             DB      00H             ;Max. value for bounds check
EXCHANGE        DB      00H             ;Swap flag for sort
                                        ;00 = no swap
                                        ;01 = swap
NAMES_DISPLAYED DB      00H             ;Number of names on the
                                        ;display.
DIR_SEC         DW      00H             ;Number of sectors to read
DIR_BEG         DW      00H             ;Beginning directory number
ENTRIES         DW      00H             ;Number of possible entries for
                                        ;drive format
DELAY_COUNT     DW      07FFH           ;Inner loop delay count
LOOP_VALUE      DW      0100H           ;Number of times to execute
                                        ;the outer loop of the delay.
MY_DATA ENDS
;**********************************************************

MY_CODE SEGMENT PARA    'CODE'
```

Listing 8-4 continued

```
        ASSUME  CS:MY_CODE,DS:MY_DATA,ES:MY_DATA,SS:MY_STACK

START   PROC    FAR
BEGIN1: PUSH    DS                  ;SAVE RETURN ADDRESS
        XOR     AX,AX               ;ZERO ACCUMULATOR
        PUSH    AX                  ;SAVE RETURN SEG. OFFSET.
        MOV     AX,MY_DATA          ;SET UP SEG REGISTERS
        MOV     DS,AX
        MOV     ES,AX
        PUSH    ES                  ;Save ES
        @SET_DTA    DIR_BUFF        ;Set the disk transfer address
        POP     ES                  ;Restore value
;******** Clear the screen and set the cursor *********************

BEGIN2: NOP
        @SCROLL 06,00,18H,4FH,00,00,07  ;Clear the screen
        NOP
        @CURSET 0,0,0               ;Set the cursor to row 0, col 0,
                                    ;screen 0.

PROMPT: NOP
        @VDLINE MESSAGE2            ;Ask user which drive.
        @WAITKEY                    ;Get the response in AL.
        AND     AL,01011111B        ;Reset bit 5, convert
                                    ;to uppercase
        MOV     MIN,41H             ;Ascii 'A' is minimum
        MOV     MAX,44H             ;Ascii 'D' is maximum
                                    ;
        CALL    CHECK_BOUNDS        ;See if response is valid.
        JNC     ALL_RIGHT           ;If carry not set then continue.
ENTRY_ERROR: NOP
        @VDLINE MESSAGE3            ;Display error message
        CALL    LONG_DELAY          ;Twiddle some thumbs, leave
                                    ;the message on the screen.
        JMP     BEGIN2              ;Restart Program.

ALL_RIGHT:      AND     AL,0FH      ;Mask high order nibble
        DEC     AL                  ;Decrement AL register.
                                    ;If Drive A, then AL =0
                                    ;B = 1, C = 2, etc.
        MOV     DRIVE,AL            ;Save drive number
        CALL    GET_DRIVE_TYPE      ;Determine the drives format
        LEA     BX,DIR_BUFF         ;Disk Transfer Address
        MOV     CX,DIR_SEC          ;Number of sectors to read
        MOV     DX,DIR_BEG          ;Beginning relative sector.
        INT     25H                 ;Perform absolute sector read.
        POPF                        ;Retrieve status.
        JNC     NO_ERR              ;If no error then continue.
        CALL    ERR_ROUTINE         ;Otherwise go to error handler
        JC      TOO_MANY            ;Too many errors, abort
        JMP     BEGIN2              ;And restart the program.

NO_ERR: CALL    DIR_SORT            ;Go sort directory.
        CALL    DIR_DSP             ;Go display it.
        @CRLF                       ;Go to the next line
QUIT_IT: JMP    SHORT   ALL_DONE ;Return to DOS exit point
TOO_MANY: NOP
        @VDLINE MESSAGE1 ;Error message.
        CALL    LONG_DELAY          ;Wait for a awhile (display mssg)
        @SCROLL 06,00,18H,4FH,00,00,07  ;Clear the screen
ALL_DONE:       RET             ;Return to DOS

START   ENDP
;**************************************************************
;Bounds checking routine. This routine will compare the
;binary value in AL to the upper and lower limits set in
;RAM locations MIN and MAX. If the routine finds the value
;out of bounds, the carry will be set. Otherwise the
;routine will return control with the carry reset.
;
CHECK_BOUNDS    PROC    NEAR
        CMP     AL,MIN          ;Is AL < MIN?
        JB      OUT_BOUNDS      ;Jump if AL is below.
        CMP     AL,MAX          ;Is AL > Max?
        JA      OUT_BOUNDS      ;Jump if AL is above max.
        CLC                     ;Otherwise, clear carry.
        JMP     SHORT CB_DONE   ;Return point
OUT_BOUNDS: STC                 ;Set the carry flag.
CB_DONE:        RET             ;Return to caller.
CHECK_BOUNDS    ENDP

;********************************************************
;This routine will provide for a software delay of variable
;length. Uses BX and CX. All registers used are restored
;prior to exiting this routine.
;
LONG_DELAY      PROC    NEAR
        PUSH    BX              ;Save registers used in routine.
        PUSH    CX              ;
        MOV     CX,DELAY_COUNT  ;Get count value
IN_LOOP1: MOV   BX,LOOP_VALUE   ;Get loop value
IN_LOOP2: DEC   BX              ;Decrement inner loop count
          JNZ   IN_LOOP2        ;Loop if BX is not zero.
          LOOP  IN_LOOP1        ;Loop if CX is not zero.
          POP   CX              ;Restore registers
          POP   BX              ;
          RET                   ;All done, return to caller.
LONG_DELAY      ENDP

;*************************************************************
;This routine is the error handler for disk access. If there
;is an error during disk access, the memory location
;ERR_COUNT is incremented. If the count is greater than
;10, the operation is aborted, and the program is restarted.
;If the count is less than ten, the operation is retried.
;The main program determines this from the state of the carry flag
;on return from this routine. If the carry is set, then the
;operation should be aborted.
;
ERR_ROUTINE     PROC    NEAR
        INC     ERR_COUNT       ;Increment memory
        CMP     ERR_COUNT,11    ;More than 10 errors?
        JAE     MORE_TEN        ;If so, then set carry
        CLC                     ;Else clear the carry
        JMP     SHORT ERR_DONE  ;Return point
MORE_TEN:       STC             ;Set the carry flag.
ERR_DONE:       RET             ;Return to caller
ERR_ROUTINE     ENDP

;***********************************************************
```

Listing 8-4 continued

```
;This routine will sort the directory read in from disk.
;The directory will reside in the memory buffer area DIR_BUFF
;after the disk read operation.
;The format of the directory is:
;Track 0, Sectors 4-7
;Bytes 0-7 Filename. If byte 0 = 00H then entry never used.
;                                E5H then entry has been deleted
;                                2EH then the entry is for a directory
;       Any other character found in byte 0 is the first
;       character of the filename.
;8-10 The file name extension.
;11   The File attribute 01 = Read Only
;                        02 = Hidden File
;                        04 = System File
;                        08 = Volume Label (in bytes 0-10)
;                        10 = Subdirectory
;                        20 = Archive Bit, set whenever
;                             the file has been written to and
;                             closed, and has not been backed up
;12-21 Reserved By MSDOS
;22-23 The time the file was last updated or created.
;      Format: B11-15 = Hour (0-23)
;              B5-10  = Minutes (0-59)
;              B0-4   = two-second increments
;       Least significant byte = Byte 22
;       Most Significant Byte  = Byte 23
;
;24-25 Date of file creation or update.
;       Format: B8-15 = Year (0-119 = 1980-2099)
;               B5-7  = Month (1-12)
;               B4-0  = Day    (1-31)
;
;26-27 Starting Cluster; See IBM DOS Technical Reference Manual
;                        Page 4-7 and Page 4-10.
;28-31 File Size in bytes (Least Significant Word = LSB of the size)
;
;The first part of the procedure extracts the directory filenames
;from the DTA which is DIR_BUFF
;and places it in DIR_BUF2.
;This is where the directory will be sorted.
;
DIR_SORT        PROC    NEAR

MOVE_DIR:       MOV     CX,ENTRIES       ;Number of entries in dir.
                                         ;move.
                LEA     DI,DIR_BUF2      ;Set destination register
                LEA     SI,DIR_BUFF      ;Source DS:SI
                CLD                      ;Clear direction flag
                                         ;for auto increment
                SUB     BX,BX            ;Zero base register
MOVE_DIR2:      CMP     DIR_BUFF[BX],00H ;Entry never used?
                JZ      GET_NEXT         ;If so then get next record
                CMP     DIR_BUFF[BX],0E5H ;Empty due to deletion?
                JZ      GET_NEXT         ;If so then get next entry
MOVE_DIR3:      PUSH    CX               ;Save number of entries
                MOV     CX,11            ;Filename and extension
        REP     MOVSB                    ;Move filename
                ADD     BX,32            ;Offset to next entry
                ADD     SI,32-11         ;Point to next entry
                POP     CX               ;Recover number of entries
                JMP     SHORT GET_NEXT2 ;Continue from here
GET_NEXT:       ADD     BX,32            ;Point to next entry
                ADD     SI,32            ;If we did not find an entry,
GET_NEXT2:      INC     ENTRY_COUNTER    ;Increment entry found counter
                LOOP    MOVE_DIR2        ;Continue for all entries
;**************                        ***************************
;Now sort the entries into alphabetical order.
;Use bubble sort technique.
;
DIR_SORT2:      MOV     CL,ENTRY_COUNTER ;Number of entries to
                MOV     CH,0              ;sort.
                MOV     EXCHANGE,00H     ;Clear the swap flag.
                LEA     DI,DIR_BUF2[11] ;Destination of compare
                LEA     SI,DIR_BUF2      ;Source for compare
                PUSH    SI               ;Store pointers on stack
                PUSH    DI
DIR_SORT3:      PUSH    CX               ;Save the number of times to
                                         ;execute this routine.
                MOV     CX,8             ;Number of bytes to compare
        REPE    CMPSB                    ;Compare entire string
                JNA     NO_SWAP          ;If source > destination
                                         ;Else, swap entries
                CALL    SWAP_ENTRY       ;Go swap entries
NO_SWAP:        CALL    ADJ_POINTERS     ;Adjust pointers
                POP     CX               ;Retrieve count
                LOOP    DIR_SORT3        ;Loop for all entries
                POP     DI               ;Clean up stack
                POP     SI
                CMP     EXCHANGE,0H      ;If not equal to zero
                                         ;then make another pass
                JNE     DIR_SORT2        ;Redo routine if there
                                         ;was an entry swapped.
                RET                      ;Otherwise return
DIR_SORT        ENDP
;*************************************************************
;Procedure to swap directory entries. Expects the stack to be
;arranged as follows:
;SP -->                                         TOP OF STACK
;      - Base Pointer (Pushed by this routine) Bytes 0 & 1
;      - Return Address 1 -                     Bytes 2 & 3
;      - Count Register -                       Bytes 4 & 5
;      - Destination Pointer -                  Bytes 6 & 7
;      - Source Pointer -                       Bytes 8 & 9
;
;      - Return Address 2 -                     Bytes 10 & 11
;
;Uses the pointers which are saved on the stack.
;
SWAP_ENTRY      PROC    NEAR
                PUSH    BP               ;Save BP
                MOV     BP,SP            ;Point to stack
                MOV     DI,[BP+6]        ;Recover pointers
                MOV     SI,[BP+8]        ;from stack
                MOV     CX,11            ;Number of bytes to exchange
SWAP_BYTES:     MOV     AL,[DI]          ;Get byte to exchange
                XCHG    AL,[SI]          ;Bytes exchanged
                MOV     [DI],AL          ;Save byte from source in dest.
                INC     SI               ;Point to next byte(s)
                INC     DI               ;
```

Listing 8-4 continued

```
                LOOP    SWAP_BYTES      ;Continue until zero.
                MOV     EXCHANGE,01     ;Show there was an exchange.
                POP     BP              ;Recover bas register
                RET                     ;Return to caller
SWAP_ENTRY      ENDP
;*****************************************************************
ADJ_POINTERS    PROC    NEAR
                PUSH    BP              ;Save base register
                MOV     BP,SP           ;Point to base of stack
                MOV     DI,[BP+6]       ;Get destination pointer
                MOV     SI,[BP+8]       ;Get source pointer
                ADD     DI,11           ;Point to next filename
                ADD     SI,11           ;Point to next filename to compare
                MOV     [BP+6],DI       ;Resave pointer
                MOV     [BP+8],SI       ;Resave pointer
                POP     BP              ;Restore base register
                RET                     ;Return to caller
ADJ_POINTERS    ENDP
;***************************************************************
;This procedure will display the directory entries stored in
;DIR_BUF2. The procedure will display up to 64 names on the
;screen at a time, and wait for the operator to press any key before
;display any remaining entries. As the double sided formats
;allow up to 112 directory entries, the pause between screen displays
;prevents the screen from scrolling before the operator can read
;the directory names.
;
DIR_DSP PROC    NEAR
        @SCROLL 06,00,18H,4FH,00,00,07  ;Clear screen
                                        ;Then set the cursor
        @CURSET 00,00,00                ;Screen 0, row 0, col 0
        MOV     CL,ENTRY_COUNTER        ;Number of entries in dir.
        INC     CL                      ;
        MOV     CH,0                    ;
        LEA     SI,DIR_BUF2             ;Get start of buffer address
DIR_DSP2: PUSH  CX                      ;Save entry counter
        MOV     CX,8                    ;Display 8 character filename
        CMP     BYTE PTR [SI],00        ;If zero then skip
        JNE     DSP_NEXT_CHAR           ;If not, then display filename
        POP     CX                      ;Retrieve # of times to execute
                                        ;this loop.
        ADD     SI,11                   ;Point to next field.
        LOOP    DIR_DSP2                ;Scan all entries
DSP_NEXT_CHAR: MOV      DL,[SI]         ;Get character in filename
        @CHARDSP                        ;Display character in DL
        INC     SI                      ;Increment source pointer
        LOOP    DSP_NEXT_CHAR           ;Loop till al chars are dsp
        MOV     DL,'.'                  ;Filename delimiter
        @CHARDSP                        ;Display '.'
        MOV     CX,3                    ;3 character extension
DSP_EXT: MOV    DL,[SI]                 ;Get character from buffer
        @CHARDSP                        ;Display the character
        INC     SI                      ;Point to the next character
        LOOP    DSP_EXT                 ;Display all 3 characters
        MOV     CX,8                    ;Display 8 spaces
        MOV     DL,20H                  ;ASCII Space
DSP_SPACE: @CHARDSP                     ;
        LOOP    DSP_SPACE               ;Do all 8 spaces
        INC     NAMES_DISPLAYED         ;Inc number of names on screen
        CMP     NAMES_DISPLAYED,80      ;80 NAMES DISPLAYED?
        JB      CONT_DSP                ;If < 80, then continue.
        @WAITKEY                        ;Wait for a key to be hit.
        MOV     NAMES_DISPLAYED,0       ;Clear filename counter
CONT_DSP: POP   CX                      ;Recover outer loop count
        LOOP    DIR_DSP2                ;Continue until all files are
                                        ;displayed
        MOV     NAMES_DISPLAYED,0       ;Clear name counter
        RET                             ;All done, go home
DIR_DSP ENDP
;*******************************************************************
;This procedure will read the FAT (track 0, sector 1) via
;MSDOS interrupt type 25H. The byte is the key for the drive type
;(i.e. single sided/double sided 8/9 sectors/track etc.)
;See the beginning of this file for the possible drive configurations
;based on the 1st byte value of the file allocation table (FAT).
GET_DRIVE_TYPE  PROC    NEAR
        PUSH    AX                      ;Save ALL REGISTERS!
        PUSH    BX
        PUSH    CX
        PUSH    DX
        LEA     BX,DIR_BUFF             ;DTA
        MOV     AL,DRIVE                ;Get drive number
        MOV     CX,01                   ;Read one sector
        MOV     DX,01                   ;AT SECTOR 1
        INT     25H                     ;Absolute disk read
        POPF                            ;Any errors?
        JNC     NO_ERR2                 ;NO ERRORS, CONTINUE
        STC                             ;Else set the carry and return
        JMP     DONE_WITH_TYPE          ;Leave this routine.
NO_ERR2:        LEA     SI,DIR_BUFF     ;Set pointer to first byte
        MOV     AL,BYTE PTR [SI]        ;What type of drive is it?
        INC     AL                      ;Increment byte, was it FFH?
        JZ      SET_DS8ST               ;0FFH = DS 8 SECTORS/TRACK
        INC     AL                      ;Was it 0FE?
        JZ      SET_SS8ST               ;0FEH = SS 8 Sectors/track
        INC     AL                      ;Was it 0FD?
        JZ      SET_DS9ST               ;0FDH = DS 9 Sectors/track
        JMP     SHORT   SET_SS9ST       ;Else it is SS 9 Sectors/track
SET_SS8ST:      MOV     DIR_SEC,4       ;Number of directory sectors
        MOV     DIR_BEG,3               ;Directory start sector
        MOV     ENTRIES,64              ;Number of possible directory
                                        ;entries.
        JMP     SHORT   DONE_WITH_TYPE  ;All done, return to caller
SET_DS8ST:      MOV     DIR_SEC,7
                MOV     DIR_BEG,3
                MOV     ENTRIES,112
                JMP     SHORT   DONE_WITH_TYPE
SET_DS9ST:      MOV     DIR_SEC,7
                MOV     DIR_BEG,5
                MOV     ENTRIES,112
                JMP     SHORT   DONE_WITH_TYPE
SET_SS9ST:      MOV     DIR_SEC,4
                MOV     DIR_BEG,5
                MOV     ENTRIES,64
                CALL    CLR_DTA         ;Clear the DTA
DONE_WITH_TYPE: POP     DX              ;Restore registers
                POP     CX
                POP     BX
                POP     AX
```

Listing 8-4 continued

```
                RET                             ;Exit this procedure
GET_DRIVE_TYPE  ENDP
;*************************************************************
CLR_DTA PROC    NEAR
        MOV     CX,LENGTH DIR_BUFF              ;Number of bytes to clear
        LEA     SI,DIR_BUFF                     ;Point to first byte of buffer
CLR_DTA2:       MOV     BYTE PTR [SI],00H ;Clear the byte
        INC     SI                              ;Increment the address pointer
        LOOP    CLR_DTA2                        ;Clear all bytes
        RET
CLR_DTA ENDP
;*******************************************************************
MY_CODE ENDS
        END     START
```

Listing 9-1
GRAPHIC.ASM

```
                PAGE    60,132
                TITLE   Graphic Routine
                SUBTTL  BY: Gary A. Shade

                .XLIST
                INCLUDE B:DOSEQU.EQU
                .LIST

                IF1
                INCLUDE B:MACFLE.MAC
                ENDIF
;*************************************************************
;Graphic ROUTINE
;Created: 09-01-84
;By: Gary A. Shade
;Date last revised: 09-02-84
;(c) 1984 Gary A. Shade. All rights reserved.
;
;This file will define a program which demonstrates the use
;of a procedure to draw lines on the IBM PC.
;As MSDOS does not offer adequate graphic routines, the
;routine found here will utilize BIOS calls. This may cause some degree
;of incompatibility to exist when these macro calls are executed on
;the so called 'compatible' computers. The routine has been tested
;on an IBM PC and a Compaq computer.
;
;*******************************************************************
;This procedure will draw a line from a specified start position,
;to the specified stop position. There is no check to see if the
;parameters supplied are within bounds (i.e. row and column min/max
;values). You must be sure they are within the range for the current
;graphics mode selected. (320x200 medium res.)
;and (640 x 200 high res).
;Use the @VDMODE macro in MACFLE.MAC to set the video mode
;as desired.
;The routine expects the following RAM locations to contain the
;values for the starting X-Y, and ending X-Y coordinates.
;
;STARTX = starting x coordinate
;STARTY =   ""     y     ""
;STOPX  = ending x coordinate
;STOPY  = ending y coordinate
;DIRECTIONX = 00H = FORWARD MOVE: IF <> 00H THEN BACKWARD MOVE
;DIRECTIONY = 00H = MOVE UP : IF <> 00H THEN MOVE DOWN.
;
;******************************************************************
;Define a macro to be used in this routine to erase an object (line)
;on the screen.
```

Listing 9-1 continued

```
;****************************************************************
ERASE   MACRO   STARTPX,STOPPX,STARTPY,STOPPY,COLORP
        MOV     STARTX,STARTPX
        MOV     STOPX,STOPPX
        MOV     STARTY,STARTPY
        MOV     STOPY,STOPPY
        MOV     COLOR,COLORP
        CALL    DRAW_LINE
        ENDM

;****************************************************************

MY_STACK        SEGMENT PARA    STACK    'STACK'
        DW      100 DUP(?)
MY_STACK        ENDS
;****************************************************************
MY_DATA SEGMENT PARA    'DATA'
STARTX  DW      320                       ;START at center screen
                                          ;use high res mode.
STOPX   DW      470                       ;stop point.
STARTY  DW      100                       ;Start of y = center screen
STOPY   DW      4                         ;Go up.
DIRECTIONX      DB      0                 ;Initialze direction as fwd.
DIRECTIONY      DB      0                 ;and up.
D_L_FLAG        DB      0                 ;flag for reaching the end
                                          ;of X and Y. Bit 0 = 1 =
                                          ;end of X reached.
                                          ;Bit 1 = 1 then end of Y
                                          ;Reached.
COLOR           DB      0                 ;Used to specify the color
                                          ;attribute of the dot.
MY_DATA ENDS
;******************************************************************
;The color combinations in this example are white on black, and
;black on black due to using the high res. mode of operation.
;you can specify other color combinations if using the
;medium resolution mode. See the text for an explanation of the
;possible modes of operation.
;******************************************************************
W_B     EQU     07H                       ;White on black
B_B     EQU     00H                       ;Black on black
;******************************************************************
;The main procedure is simply a test program for the
;graphic procedure DRAW_LINE. The programmer must define the
;start and stop positions of the X and Y axis. The coordinates
;must be within the capabilities of the graphics mode. For
;medium resolution displays this means an X value from 0-319,
;and a Y value of 0-199. In high resolution mode, the X values
;must lie in the range of 0-639, and must contain Y values
;between 0-199. The graphic routines do not perform bounds
;checking on the values supplied!
;
;This program will draw a large 'A' on the screen and
;create a moving border to create the effect of animation.
;
MY_CODE SEGMENT PARA    'CODE'
        ASSUME  CS:MY_CODE,DS:MY_DATA,ES:MY_DATA,SS:MY_STACK
MAIN_PROG       PROC    FAR
```

```
START:  PUSH    DS                      ;Save code segment
        XOR     AX,AX                   ;Zero offset return
        PUSH    AX                      ;Save offset
        MOV     AX,MY_DATA              ;Set upsegment registers
        MOV     DS,AX
        MOV     ES,AX
;****** Get the current video mode, save it, and establish the
;****** new video mode. For this example, use mode 6; the high
;****** resolution mode.
        @VDMODERD                       ;Get the current status
        PUSH    AX                      ;Save the video mode
                                        ;which was returned in AL
        MOV     CX,40                   ;Do the procedure 40 times.
        @VDMODE 6                       ;Set high res mode.
;****** Start of the test program ******************************
;****** Draw the letter 'A' on the screen **********************

DRAW_IT: MOV    STARTX,320              ;Set parameters to draw
        MOV     STOPX,470               ;End point
        MOV     STARTY,50               ;Now define Y axis
        MOV     STOPY,150
        MOV     COLOR,W_B               ;White on black
        CALL    DRAW_LINE               ;Go draw the line defined.
DRAW_IT2: MOV   STARTX,320              ;Define next line.
        MOV     STOPX,170
        MOV     STARTY,50               ;Define Y axis
        MOV     STOPY,150
        MOV     COLOR,W_B
        CALL    DRAW_LINE
        CALL    DELAY                   ;Delay a while
DRAW_IT3: MOV   STARTX,275              ;Horizontal line
        MOV     STOPX,375               ;Bar for the 'A'.
        MOV     STARTY,100
        MOV     STOPY,100
        MOV     COLOR,W_B               ;WHITE ON BLACK
        CALL    DRAW_LINE               ;Draw it.
        CALL    DELAY                   ;Delay awhile
        CALL    DELAY
;******* Now draw a border around the letter. After a short delay,
;******* erase the line, and redraw the shape elsewhere on the screen.
;This is known as animation.
;
DRAW_2: MOV     STARTX,0                ;Draw a straight line
        MOV     STOPX,540               ;
        MOV     STARTY,199              ;
        MOV     STOPY,199               ;
        MOV     COLOR,W_B               ;
        CALL    DRAW_LINE               ;Draw the line.
        CALL    DELAY                   ;Wait awhile
        ERASE   0,540,199,199,B_B       ;Erase the line.
        CALL    DELAY                   ;delay a bit.
DRAW_3: MOV     STARTX,540              ;draw a new line
        MOV     STOPX,540               ;from bottom up on right of screen
        MOV     STARTY,199
        MOV     STOPY,0
        MOV     COLOR,W_B               ;White on black
        CALL    DRAW_LINE               ;draw the line
        CALL    DELAY                   ;delay awhile
        ERASE   540,540,199,0,B_B       ;and erase the line.
```

Listing 9-1 continued

```
        CALL    DELAY
DRAW_4: MOV     STARTX,540              ;Draw line from top right
        MOV     STOPX,0                 ;to top left of screen
        MOV     STARTY,0
        MOV     STOPY,0
        MOV     COLOR,W_B
        CALL    DRAW_LINE
        CALL    DELAY
        ERASE   540,0,0,0,B_B           ;Now erase it.
        CALL    DELAY
DRAW_5: MOV     STARTX,1                ;draw line from top to bottom on lef
        MOV     STOPX,1                 ;of the screen
        MOV     STARTY,0
        MOV     STOPY,199
        MOV     COLOR,W_B
        CALL    DRAW_LINE
        CALL    DELAY
        ERASE   1,1,0,199,B_B           ;And erase this line.
        CALL    DELAY

        DEC     CX                      ;Decrement counter
        JNZ     DRAW_IT_ISLAND          ;Jump to the 'jump' island.

        @WAITKEY                        ;Wait for a key closure before
                                        ;ending.
        POP     BX                      ;Restore old video mode
        @VDMODE BL                      ;Old mode is in BL
        RET                             ;All done - return to MSDOS

DRAW_IT_ISLAND: JMP     DRAW_2          ;This is an island jump point
                                        ;to jump to the start of program.
                                        ;see the text for an explanation
                                        ;of why islands are sometimes
                                        ;needed.
MAIN_PROG       ENDP
;*******************************************************************
;This is the line drawing routine which will draw a line on the
;selected graphics screen by setting a range of dots specified
;by start and stop, X and Y values. The routine automatically
;determines if the dots must be drawn in a forward or reverse,
;or down or up direction. Simply supply the start and stop
;X and Y values in the RAM locations, STARTX, STOPX, STARTY, and STOPY.
;The routine also needs scratch pad RAM for DIRECTIONX, DIRECTIONY,
;and storage for a flag byte D_L_FLAG which is used for X, Y
;graphic control.

DRAW_LINE       PROC    NEAR
        PUSH    CX                      ;Save count.
        MOV     AX,STOPX                ;Calculate direction
        SUB     AX,STARTX               ;If no carry then move forward
        JNC     FORWARDX
        MOV     DIRECTIONX,01           ;Direction = 01 = backwards
        JMP     SHORT D_L_2             ;And continue past next part
                                        ;of code.
FORWARDX:   MOV DIRECTIONX,0H           ;
D_L_2:  MOV     AX,STOPY                ;Calculate direction of Y
                                        ;(up or down)
        SUB     AX,STARTY               ;IF STOP > START THEN MOVE
                                        ;DOWN
        JNC     DOWNY
        MOV     DIRECTIONY,0H           ;Move down
        JMP     SHORT D_L_3             ;AND CONTINUE
DOWNY:  MOV     DIRECTIONY,01H          ;01 means move down.

D_L_3:
        @SET    STARTY,STARTX,COLOR     ;ROW, COLUMN AND COLOR
        CMP     D_L_FLAG,03H            ;See if bit 0 and 1 set
        JE      D_L_4                   ;If set, exit
        MOV     AX,STARTX               ;Compare to end
        CMP     AX,STOPX                ;If equal set flag
        JNE     D_L_5                   ;If not, then skip.
        OR      D_L_FLAG,01H            ;Set the flag
        JMP     SHORT   B_2             ;Go check Y
D_L_5:  CMP     DIRECTIONX,00H          ;Which way are we going?
        JNE     BACKWARD_1              ;Draw backwards
        INC     STARTX                  ;increment column
        JMP     SHORT   B_2             ;AND CONTINUE
BACKWARD_1: DEC STARTX                  ;Decrement column
B_2:
        MOV     AX,STARTY               ;are we at the end of rows?
        CMP     AX,STOPY                ;
        JNE     D_L_6                   ;If not do not set flag.
        OR      D_L_FLAG,02H            ;Set bit 2
        JMP     SHORT U_2               ;Skip inc. or dec. of Y
D_L_6:  CMP     DIRECTIONY,00H          ;Which way for Y?
                                        ;00H means up.
        JZ      UP_1
        INC     STARTY                  ;Increment row
        JMP     SHORT   U_2             ;AND CONTINUE
UP_1:
        DEC     STARTY                  ;Decrement Y position
U_2:
        JMP     D_L_3                   ;Repeat operation
D_L_4:
        MOV     DIRECTIONX,00           ;Reset routine directives
        MOV     DIRECTIONY,00
        MOV     D_L_FLAG,0
        POP     CX                      ;Retrieve count.
        RET                             ;Exit procedure
DRAW_LINE       ENDP
;*********************************************************
;This routine generates a fixed delay.
DELAY   PROC    NEAR
        PUSH    CX              ;Save register
        MOV     CX,50           ;COUNT = 50 for delay loop.
DELAY_LOOP:     LOOP    DELAY_LOOP ;Approx delay = 50 * .01 Sec. = 1/2 sec
        POP     CX              ;Restore register
        RET
DELAY   ENDP
;******************************************************************
```

Listing 9-2
SOUND.ASM

```
        PAGE    60,132
        TITLE   SOUND GENERATION
        SUBTTL  A SOUND PROGRAM

        .XLIST
        INCLUDE B:DOSEQU.EQU
        .LIST

        IF1
        INCLUDE B:MACFLE.MAC
        ENDIF

;*********************************************************
;Sound Procedure
;Created: 06-14-84
;By: Gary A. Shade
;Last Revised: 08-12-84
;(c) 1984 Gary A. Shade.  All Rights Reserved
;
;
;This program will produce a specified tone, of a specified
;duration from the IBM PC's internal speaker.
;It must directly address the 8255A which is the interface chip
;that controls the speaker. Bits 0, and 1 of the Programmable
;Peripheral Interface (PPI), turn the speaker on and off.
;Bit 1 directly drives the speaker, while bit 0 gates the output
;of the 8253 programmable timer to the speaker.
;The clock input to the timer is 1.19 MHz, and the timer provides
;a system clock tick 18.2 times/second should you want to program
;the 8253 timer directly. See the IBM Technical Reference Manual
;(# 6025005) for more details.
;
;
;The PPI is located at the following port addresses:
;60H -> Input port used for keyboard scans
;61H -> B0 = Speaker gate from timer 2 of 8253
;       B1 = Speaker data
;62H -> Input used for system functions see IBM Technical
;       Reference Manual (#6025005) page I-12 for more details.
;
;************************************************************
;Define speaker output port
SPEAKER EQU     61H
;************************************************************
;
;Define
MY_STACK        SEGMENT PARA    STACK   'STACK'
        DW      100 DUP(?)
```

```
MY_STACK        ENDS
;*****************************************************************
MY_DATA SEGMENT PARA     'DATA'
FREQUENCY_TABLE DW       100,3 DUP (530,662,694,662),3 DUP(503)
                DW       2 DUP(694,694,694,530,592,592)
                DW       2 DUP(530,662,694,662),4 DUP(530)
                DW       2 DUP(494,494,330,434,492)
                DW       00                        ;END OF TABLE

DURATION_TABLE  DW       15 DUP(25),50
                DW       4 DUP(25,25,50)
                DW       10 DUP(25),100,200
                DW       10 DUP(64)
DURATION        DW       0                         ;Used in sound routine
FREQUENCY       DW       0
MESSAGE1        DB       'Hit any key to begin SOUND! ','$'

MY_DATA ENDS
;*****************************************************************
MY_CODE SEGMENT PARA     'CODE'
SOUNDS  PROC    FAR
        ASSUME  CS:MY_CODE,DS:MY_DATA,ES:MY_DATA,SS:MY_STACK
BEGIN_SOUND: PUSH DS                     ;SAVE MSDOS RETURN SEGMENT
        XOR     AX,AX                    ;ZERO AX AND SAVE OFFSET
        PUSH    AX                       ;
        MOV     AX,MY_DATA               ;SET UP SEGMENT REGISTERS
        MOV     DS,AX
        MOV     ES,AX
        XOR     SI,SI                    ;
        XOR     DI,DI                    ;Zero pointers

PROMPT:                                  ;Prompt user to hit any key.
        @SCROLL 06,00,18H,4FH,00,00,07   ;Clear the screen

        @VDLINE MESSAGE1
        @WAITKEY                         ;Wait for a key to be pressed.


NEXT_NOTE: MOV  AX,FREQUENCY_TABLE[SI]   ;Set up RAM for sound routine
        MOV     FREQUENCY,AX             ;
        MOV     AX,DURATION_TABLE[DI]    ;Set duration of note
        MOV     DURATION,AX
        CALL    SOUND_GEN                ;CALL THE SOUND GENERATOR
        INC     SI                       ;Point to the next entry
        INC     SI
        INC     DI                       ;
        INC     DI
        CMP     FREQUENCY_TABLE[SI],00H  ;If 00 encountered, then
                                         ;the value is a stop code
        JNE     NEXT_NOTE                ;Play another note.
        RET
SOUNDS  ENDP                             ;End of main procedure
;
;*****************************************************************
;Sound generator procedure. Must have the memory locations
;sound, and frequency set with  the proper values before
;calling this procedure. All registers are saved on entry, and
;restored on exit from this routine.
;
;
SOUND_GEN       PROC     NEAR
```

Listing 9-2 continued

```
        PUSH    AX                      ;Save registers
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     BX,DURATION             ;Set up duration value
        IN      AL,SPEAKER              ;Get current value and
                                        ;save it for later.
        PUSH    AX                      ;
        CLI                             ;Disable interrupts
GENERATE:       AND     AL,0FCH         ;Reset bit 1 & 0 of PPI
        OUT     SPEAKER,AL
GEN_COUNT: MOV  CX,FREQUENCY            ;Defines inner loop count
FREQ_OUT:       LOOP    FREQ_OUT        ;Inner delay loop = 10 ms
        OR      AL,03H                  ;Turn speaker off.
        OUT     SPEAKER,AL
        MOV     CX,FREQUENCY            ;Keep it off for a while
FREQ_OUT_2:     LOOP    FREQ_OUT_2      ;Loop value = 10 ms.

        DEC     BX                      ;Decrement outer loop count
        JNZ     GENERATE                ;If not done, repeat sequence
        POP     AX                      ;Get old value of the port
        OUT     SPEAKER,AL              ;Restore original port values
        STI                             ;Restore interrupts
        POP     DX                      ;Restore registers
        POP     CX
        POP     BX
        POP     AX
        RET                             ;Return to caller
SOUND_GEN       ENDP                    ;End of this procedure
;*****************************************************************
MY_CODE ENDS                            ;End of segment
        END     BEGIN_SOUND             ;Define program entry point
```

Listing 10-1
DLOAD.ASM

```
                        PAGE 60.132
                        TITLE DLOAD.ASM
                        SUBTTL FILE DOWNLOAD EXAMPLE

                        .XLIST
                        INCLUDE B:DOSEQU.EQU
                        .LIST

                        IF1
                        INCLUDE B:MACFLE.MAC
                        ENDIF

;************************************************************
;File download example
;By: Gary A. Shade
;Date: 07-10-84
;Last Updated: 03-07-85
;(c) All rights reserved.
;This program will sequentially read a text file from disk.
;This program will work with MSDOS 2.0.
;It is to be used in conjunction with
;the telecommunications program module: COMM.ASM. The program
;module here routes all disk output to the TX_BUFFER defined in
;the telecom module.
;See the external list for a list of the labels and messages which
;must be defined int he telecommunications module as PUBLIC.
;
;The procedure terminates when an end of file character is
;encountered (HEX 1AH) when reading from the disk file.
;
;
;
 EXTRN DISK_BUFFER_S:BYTE,TX_BUFFER_IN:WORD,TX_BUFFER_OUT:WORD
 EXTRN TX_CHARS:BYTE,TX_BUFFER:BYTE
 EXTRN SYSTEM_STATUS_S:BYTE,FILE_HANDLE_S:WORD,MESSAGE17:BYTE
 EXTRN MESSAGE20:BYTE,MESSAGE42:BYTE
 EXTRN MESSAGE41:BYTE,MAX_CHARS:BYTE
 EXTRN KBUFFER:BYTE,CHARS_TYPED:BYTE
 EXTRN KBDBUFFER:BYTE,MESSAGE20:BYTE
 EXTRN MESSAGE45:BYTE,MESSAGE46:BYTE,MESSAGE44:BYTE
 EXTRN MESSAGE47:BYTE


 PUBLIC START_TX_FILE

MY_CODE SEGMENT PARA PUBLIC  'CODE'
```

Listing 10-1 continued

```
START_TX_FILE   PROC     FAR
        ASSUME  CS:MY_CODE

START_1:
        TEST     SYSTEM_STATUS_S,10000000B   ;See if send bit
                                             ;already set.
                                             ;Skip file name fetch
                                             ;and open routine if it
                                             ;is.
        JZ       START_2
        JMP      READ_FILE


START_2:
        MOV      TX_BUFFER_IN,0            ;Reset tx buffer pointers
        MOV      TX_BUFFER_OUT,0           ;
        MOV      TX_CHARS,0

        @SCROLL 06,00,18H,4FH,00,00,07     ;Clear the screen
        @VDLINE MESSAGE17                  ;Ask for the filename
        @CRLF                              ;Go to the next line.
        MOV      MAX_CHARS,40D             ;15 Characters maximum
                                           ;as in B:TESTFILE.TXT<CR>
GET_NAME:
        @KBDLINE         KBUFFER           ;Get the filename.

        XOR      BH,BH                     ;Zero high order of BX
                                           ;register since the
                                           ;offset to the first char
                                           ;typed is zero.
        MOV      BL,CHARS_TYPED
        MOV      KBDBUFFER[BX],00H         ;Terminate ASCII filename
                                           ;with a zero (NUL) byte.
        @OPEN_FILE_2 KBDBUFFER,00H         ;Macro call to open file
        JNC      G_N_2                     ;Skip if no errors
        JMP      OPEN_ERROR                ;If the carry is set then
                                           ; File open error
G_N_2:
        MOV      FILE_HANDLE_S,AX          ;Save the file handle


SET_UP:
        @SET_DTA DISK_BUFFER_S             ;Set the Disk transfer
                                           ;address
        OR       SYSTEM_STATUS_S,10000000B        ;Set high bit to
                                           ;indicate file open for
                                           ;reading.

READ_RECORD:

        @CRLF
        @VDLINE MESSAGE20                  ;File open message

READ_FILE:
        MOV      DL,0FFH                   ;See if a character is
                                           ;ready
        @CON_IO
        JZ       R_F_0                     ;Z flag set if char. not
                                           ;ready
        JMP     EOF_FOUND                   ;Else abort if a key was
                                            ;pressed
R_F_0:
        CMP     TX_CHARS,0                  ;Are all chars transmitted?
        JZ      R_F_1

        JMP     EXIT_2                      ;If not then return
                                            ;without reading.
R_F_1:
        @READ_FILE_2 FILE_HANDLE_S,DISK_BUFFER_S,255D
        JC      READ_ERROR                  ;If carry set then a
                                            ;read error
        CMP     AX,00H                      ;If zero then EOF
        JNZ     CONTINUE                    ;Else continue
        JMP     EOF_FOUND

READ_ERROR:
        CMP     AL,05H                      ;If AL = 05 then
                                            ;ACCESS DENIED
        JNZ     CHK_HANDLE                  ;Take action.
ACCESS_DENIED:
        @CRLF                               ;New line
        @VDLINE MESSAGE45                   ;Access denied
        @CRLF
        JMP     SHORT   CONT_ERROR
CHK_HANDLE:
        CMP     AL,06H                      ;If AL = 06 then
                                            ;invalid file handle
        JNZ     CONT_ERROR                  ;If not then continue
        @CRLF
        @VDLINE MESSAGE41                   ;Invalid handle
        @CRLF
CONT_ERROR:
        JMP     EOF_FOUND

CONTINUE:
        MOV     CX,AX                       ;Number of bytes to move
C_1:
        LEA        SI,DISK_BUFFER_S         ;Move from disk buffer to
                                            ;transmit buffer.
        LEA     DI,TX_BUFFER                ;
        CLD                                 ;Set for auto inc.
C_2:

        REP     MOVSB

        MOV     TX_CHARS,AL                 ;Set the number of
                                            ;characters in the buffer
        MOV     TX_BUFFER_IN,0              ;Zero pointers
        MOV     TX_BUFFER_OUT,0
        JMP     EXIT_2

OPEN_ERROR:
        CMP     AX,02H                      ;File not found?
        JNZ     O_E_1                       ;NO? Then continue
        @CRLF
        @VDLINE MESSAGE42
        @CRLF
        JMP     EXIT_2
```

Listing 10-1 continued

```
O_E_1:
        CMP     AX,04H                  ;Too many open files
        JNZ     O_E_2
        @CRLF
        @VDLINE MESSAGE44
        @CRLF
        JMP     EXIT_2
O_E_2:
        CMP     AX,05H                  ;Access denied
        JNZ     O_E_3
        @CRLF
        @VDLINE MESSAGE45
        @CRLF
        JMP     EXIT_2
O_E_3:
        CMP     AX,12H                  ;Invalid access code?
        JNZ     O_E_4                   ;Else undefined error
        @CRLF
        @VDLINE MESSAGE46
        @CRLF
        JMP EXIT_2
O_E_4:
        @CRLF                           ;Undefined error trap.
        @VDLINE MESSAGE47
        @CRLF
        JMP     EXIT_2

EOF_FOUND:
        @CRLF
        @VDLINE MESSAGE42               ;Display end of program
                                        ;message.
        @CRLF
        AND     SYSTEM_STATUS_S,00111111B ;Reset bit 7 & 6 to
                                          ;show that
                                          ;transmissions are
                                          ;inactive.
        MOV     TX_BUFFER_IN,0            ;Zero buffer pointers
        MOV     TX_BUFFER_OUT,0
        MOV     TX_CHARS,0

EXIT:
        @CLOSE_FILE_2 FILE_HANDLE_S     ;Close the file.

EXIT_2:
        RET
START_TX_FILE   ENDP                    ;End of this procedure

;************************************************************
MY_CODE ENDS
        END     START_TX_FILE
```

Listing 10-2
COMM.ASM

```
                        PAGE 60,132
                TITLE Communications program
                SUBTTL By: Gary A. Shade

                .XLIST
                INCLUDE B:DOSEQU.EQU
                .LIST

                IF1
                INCLUDE B:MACFLE.MAC
                ENDIF

;*******************************************************************
;Communications Module
;Created 08-03-84
;By: Gary A. Shade
;(C) 1984 By Gary A. Shade, All rights reserved.
;Last Revised: 2-25-85
;
;This program allows automatic buffering of incoming data to
;disk and to the printer. It uses XON/XOFF (DC1 and DC3)
;handshake protocols for flow control.
;After initializing the communications parameters, the
;program will then allow the opening of a disk file via ALT - D.
;The printer can be toggled on and off via a control-R
;and control-T sequence. To exit the program use ALT - Z.
;To see a 'help' menu use ALT - H.
;To transmit a file use ALT - B.
;
;
 PUBLIC DISK_BUFFER_S
 PUBLIC TX_BUFFER_IN
 PUBLIC TX_BUFFER_OUT
 PUBLIC TX_CHARS
 PUBLIC TX_BUFFER
 PUBLIC SYSTEM_STATUS_S
 PUBLIC FILE_HANDLE_S
 PUBLIC MESSAGE17
 PUBLIC MESSAGE20
 PUBLIC MESSAGE42
 PUBLIC MESSAGE41
```

Listing 10-2 continued

```
PUBLIC MESSAGE45
PUBLIC MESSAGE46
PUBLIC MESSAGE44
PUBLIC MESSAGE47
PUBLIC KBUFFER
PUBLIC MAX_CHARS
PUBLIC CHARS_TYPED
PUBLIC KBOBUFFER


EXTRN START_TX_FILE:FAR


;This program module will allow the user to configure the
;communications channel (serial I/O), as to baud rate, parity,
;and other parameters.
;To set the communications parameters set AL to the desired
;bit pattern as follows:
;b7  b6  b5  b4  b3  b2  b1  b0
;----------  ------      ------
;   :          :     :     :----->  Word length: 0 0 = 5 bits
;   :          :     :                           0 1 = 6 bits
;   :          :     :                           1 0 = 7 bits
;   :          :     :                           1 1 = 8 bits
;   :          :     :------------> Stop bits:   0 = 1 stop bit.
;   :          :                                 1 = 2 stop bits.*
;   :          :------------------> Parity   :   0 0 = No parity
;   :                                            0 1 = Odd
;   :                                            1 0 = No Parity
;   :                                            1 1 = Even
;   :----------------------------> Baud Rate:
;                                     0 0 0  = 110 Baud
;                                     0 0 1  = 150 Baud
;                                     0 1 0  = 300 Baud
;                                     0 1 1  = 600 Baud
;                                     1 0 0  = 1200 Baud
;                                     1 0 1  = 2400 Baud
;                                     1 1 0  = 4800 Baud
;                                     1 1 1  = 9600 Baud
;
;* = If 5 bit word length selected, and b2 = 1, then stop bits
;will be 1.5 per word transmitted. (BAUDOT and TTY).
;Use BIOS interrupt 14H to set the communications parameters.
;With AH set to function number 00
;
;Use BIOS interrupt 14H, Function number 01 to transmit a
;                                              character.
; ""    ""      ""              "" ""    02 to receive a
;                                              character.
;And BIOS interrupt 14H, Function  number 03 to read the
;communication's port status.
```

```
;
;The serial port and modem status are returned as follows:
;Function status is returned in AX as:
;       AH                                                    AL
;   Line Status                                    ; Modem Status
;------------------------------------------------------------------
;b7 = Time out                                     ; Carrier detect (CD)
;b6 = Transmitter shift register empty             ; Ring Indicator (RI)
;b5 = Transmitter holding register is empty        ; Data Set Ready (DSR)
;b4 = Break Detect                                 ; Clear to send  (CTS)
;b3 = Framing Error                                ; Delta CD
;b2 = Parity error                                 ; Trailing edge RI
;b1 = Overrun error                                ; Delta DSR
;b0 = Data Ready                                   ; Delta CTS
;-------------------------------------------------------------------
;


;
;******************************************************************
;Define stack segment

MY_STACK        SEGMENT PARA    STACK   'STACK'
        DW      256 DUP(?)                      ;256 word stack
MY_STACK        ENDS

;*******************************************************************

;*******************************************************************
;Define the data segment.
MY_DATA SEGMENT PARA    PUBLIC 'DATA'

KBUFFER LABEL   BYTE
MAX_CHARS       DB      42D     ;42 Characters maximum
CHARS_TYPED     DB      ?       ;Characters entered
KBDBUFFER       DB      42D DUP(00) ;32 byte buffer

;**** Define messages used to prompt user etc.
MESSAGE1        DB      'Enter the desired Baud Rate  ','$'
MESSAGE2        DB      '(1) = 110 Baud         (2) = 150 Baud '
                DB      '$'
MESSAGE3        DB      '(3) = 300 Baud         (4) = 600 Baud '
                DB      '$'
MESSAGE4        DB      '(5) = 1200 Baud        (6) = 2400 Baud ','$'
MESSAGE5        DB      '(7) = 4800 Baud        (8) = 9600 Baud ','$'
MESSAGE6        DB      'Use parity? (Y/N) ','$'
MESSAGE7        DB      '(1) Even               (2) Odd ','$'
MESSAGE8        DB      'How many stop bits? ','$'
MESSAGE9        DB      '(1) = 1 Stop bit       (2) = 2 Stop bits','$'
MESSAGE10       DB      'Word length? ','$'
MESSAGE11       DB      '(1) 7 Bits     (2) 8 Bits ','$'
MESSAGE12       DB      '(5 Bit and 6 bit transmission are not supported)'
                DB      '$'
```

Listing 10-2 continued

```
MESSAGE13       DB      'Communications port has been initialized ','$'
MESSAGE14       DB      'Terminal program for IBM PC (c) 1984 Gary Shade'
                DB      '$'
MESSAGE15       DB      'Disk error! - Closing file','$'
MESSAGE16       DB      'Printer is not selected! Print spooling is off'
                DB      '$'
MESSAGE17       DB      'Please enter the pathname for the file ','$'
MESSAGE18       DB      '**** Printer is ON *****',0dh,0ah,'$'
MESSAGE19       DB      '**** Printer is OFF ****',0dh,0ah,'$'
MESSAGE20       DB      '**** Disk file is now open ****',0dh,0ah,'$'
MESSAGE21       DB      '**** Disk file is now closed ****',0dh,0ah,'$'
MESSAGE30       DB      'Local echo all characters transmitted?  (Y/N) ','$'
MESSAGE31       DB      'Do a Carriage return linefeed on receipt of a CR? '
                DB      '(Y/N)','$'
MESSAGE41       DB      '** Invalid File Handle - Returning to terminal mode '
                DB      0DH,0AH,'$'
MESSAGE42       DB      'End of File found - Returning to terminal mode '
                DB      0DH,0AH,'$'
MESSAGE44       DB      'Too many files open - returning to terminal mode '
                DB      0DH,0AH,07H,'$'
MESSAGE45       DB      'File Access Denied! ',0DH,0AH,'$'
MESSAGE46       DB      'Invalid access code ',0DH,0AH,'$'
MESSAGE47       DB      'Undefined error code ',0DH,0AH,'$'
HMES1           DB      'ALT - B = Transmit a disk file'
                DB      0DH,0AH,'$'
HMES2           DB      'ALT - D = Open disk file for spooling'
                DB      0DH,0AH,'$'
HMES3           DB      'ALT - H = Display help menu'
                DB      0DH,0AH,'$'

HMES4           DB      'ALT - Z = Exit the program'
                DB      0DH,0AH,'$'
HMES5           DB      'DC1/DC3 = XON/XOFF',0dh,0ah
                DB      'DC2/DC4 = Printer ON/OFF',0dh,0ah
                db      '$'
HMES6           DB      'ALT - A = Display directory'
                DB      0DH,0AH,'$'
HMES7           DB      'ALT - X = XMODEM file transfer'
                DB      0DH,0AH,'$'
HMES8           DB      'ALT - S = Set/Change terminal characteristics'
                DB      0DH,0AH,'$'
EXIT_FLAG       DB      00          ;Flag to signal an end to the program
                                    ;If set to 01 then return to MSDOS
RS_IN_POINT     DW      00          ;Input pointer to the RS232 buffer
RS_OUT_POINT    DW      00          ;Output pointer to the RS232 buffer
RS_CHARS        DW      00          ;Number of characters in RS buffer
PRINTER_IN      DW      00          ;Printer buffer input pointer
PRINTER OUT     DW      00          ;Printer buffer output pointer
PRINTER_CHARS   DB      00          ;Number of characters in buffer
DISK_IN         DW      00          ;POINTER INTO THE DISK BUFFER
```

```
TX_CHARS        DB      00      ;Characters in buffer
TX_BUFFER_OUT   DW      00      ;Buffer output pointer
TX_BUFFER_IN    DW      00      ;Buffer input pointer
MIN             DB      00      ;Minimum value used in MINMAX routine.
MAX             DB      00      ;Maximum value used by minmax routine.
DISK_STATUS     DB      00      ;If zero then we are DC1'd for disk I/O.
                                ;If non-zero then we have sent a DC3 to the
                                ;other terminal to indicate we are near or
                                ;at a buffer full condition.
SYSTEM_STATUS   DB      00      ;Bit mapped status as follows

;       b0 = printer status: 0 = printer off
;                            1 = printer on
;       b1 = disk spooler:   0 = off, 1 = on
;       b2 = printer flow control: 0 = we are not DC3'd
;                                  1 = we are DC3'd
;       b3 = Receive DC3 status:   0 = we are not DC3'd
;                                  1 = we are DC3'd
;       b4 = receive buffer status 0 = we are not DC3'd
;                                  1 = we are DC3'd
;
;       b5 = self sent DC3         0 = we have not sent a DC3
;                                  1 = we have sent a DC3 from the keyboard.
;       b6 = local echo of characters transmitted. (True half duplex)
;            if this bit = 1 then echo.
;
;       b7 = newline function = on receipt of a carriage return ODH,
;                               do a carriage return/linefeed combination.
;                               (Only if this bit is set. If not do CR only)
;
;
;

SYSTEM_STATUS_S DB      0               ;Down load status byte
CONFIGURATION   DB      0               ;Configuration for RS-232
FILE_HANDLE     DW      0               ;File handle of an open file
                                        ;stored here. (Receive)
FILE_HANDLE_S   DW      0               ;File handle of file
                                        ;being transmitted here.
DISK_BUFFER_S   DB      512 DUP(0)      ;Buffer for transmit.



RS_BUFFER       DB      07FFH DUP(0)    ;RS-232 buffer
PRINTER_BUFFER  DB      256 DUP(0)      ;Printer buffer
DISK_BUFFER     DB      512 DUP(0)      ;Disk buffer
TX_BUFFER       DB      256 DUP(0)      ;Transmit buffer

OLD_VECTOR      DD      0               ;Storage for old RS232
                                        ;interrupt vector
```

Listing 10-2 continued

```
MY_DATA ENDS
;**********************************************************************
;Define equates used in this file.
;
DC1     EQU     11H                         ;ASCII CONTROLS
DC2     EQU     12H
DC3     EQU     13H
DC4     EQU     14H
SOH     EQU     01H

SECTOR  EQU     512                         ;512 byte sectors
SERIAL_PORT EQU 00                          ;Use serial port 1
;
;****** Uart register absolute addresses.
;THR and RHR selected for output and input if bit 7 of the
;control register = 0.
;THR is output and RHR is input.

THR     EQU     03F8H                       ;Transmitter Holding Register
                                            ;for the UART
RHR     EQU     03F8H                       ;Receiver Holding Register

;BAUD_RATE_LSB, and BAUD_RATE_MSB are selected if bit 7
;of the control register = 1
;Both are output ports
;Baud rate      MSB     LSB
;50             09H     00H
;75             06H     00H
;110            04H     17H
;134.5          03H     59H
;150            03H     00H
;300            01H     80H
;600            00H     0C0H
;1200           00H     60H
;1800           00H     40H
;2000           00H     03AH
;2400           00H     20H
;3600           00H     18H
;4800           00H     18H
;7200           00H     10H
;9600           00H     0CH

BAUD_RATE_LSB EQU       03F8H               ;Baud rate divisor. Selected
BAUD_RATE_MSB   EQU     03F9H               ;MSB of baud rate divisor.

;The Interrupt mask register is selected if bit 7 of the control
;register = 0. Otherwise BAUD_RATE_MSB above is selected.
;Bits 7-4 must be zero.
;Bit 3 = Interrupt on delta modem status
;Bit 2 = Interrupt on break received, or recieve error
```

```
;Bit 1 = Interrupt when THR is empty.
;Bit 0 = Interrupt on received character ready.
;If any of the bits above are set to one, the interrupt type is enabled.
;A zero bit means the interrupt type is disabled.

INTERRUPT_ENABLE EQU    03F9H           ;Interrupt mask register

;The interrupt ID register is a read only register and contains
;the source of the interrupt.
;If bit 0 = 1 then there is an interrupt pending.
;Bits 1 and 2 contain the interrupt type, coded as follows:
; Bit   2       1
;      ---     ---
;       0       0       Delta Modem Status
;       0       1       THR empty
;       1       0       RHR full
;       1       1       Rx error, or break detected
;

INTERRUPT_ID    EQU     03FAH           ;Interrupt source

;The line control register specifies the parameters to be used
;in the communications channel.
;Word Length
;bits 1     0
;    ---   ---
;     0     0    5 bit word length
;     0     1    6 bit word length
;     1     0    7 bit word length
;     1     1    8 bit word length
;
;Stop bits.
;
;bit 2 = 0 = one stop bit
;      = 1 = two stop bits if word length = 6, 7, or 8 bits.
;            one and one-half stop bits if word length is = 5 bits.
;Parity
;bit 3 = 0 = No parity (disabled)
;        1 = Parity enabled
;
;Parity type
;bit 4 = 0 = odd parity
;        1 = even parity
;
;Force parity bit
;bit 5 = 1 = Force spacing parity (bit 3 must = 1 and b4 =1)
;            Force marking parity (bit 3 must = 1 and b4 =0)
;
;Transmit break
;bit 6 = 1 = Force break on TX line
;        0 = disable break on TX line
;
```

Listing 10-2 continued

```
;bit 7 = 1 = Access for baud rate divisor latch
;        0 = Acess for THR and Interrupt enable register
;

LINE_CONTROL    EQU     03FBH

;The modem control register is bit mapped as follows:
;bits 7 - 5 must be a zero
;
;bit 4 = 1 = Loop back, diagnostic mode
;bit 3 = 1 = Auxillary user defined output.
;bit 2 = 1 = Auxillary user defined output.
;bit 1 = 1 = RTS = logic 0 (Space = circuit on)
;        0 = RTS = logic 1 (Mark = circuit off)
;bit 0 = 1 = DTR = locic 0
;        0 = DTR = logic 1

MODEM_CONTROL   EQU     03FCH

;The line status register returns the status of the communications
;channel as depicted in the discussion earlier, about BIOS
;call 14H which returns the status of both the modem and line in AX.
;The line status port, contains only the line status as depicted
;in the previous discussion.
;The modem status register contains only the information
;pertinent to the modem status (also discussed previously).

LINE_STATUS     EQU     03FDH
MODEM_STATUS    EQU     03FEH

;***************
;8259 INTERRUPT CONTROL PORTS.

INT_CONTROL     EQU     21H     ;Interrupt mask register
INT_COMMAND     EQU     20H     ;Command register

;*********************************************************************
;Define code segment. The procedure MAIN_PROG, is the main program
;loop used for this communications program.
;It will set a new interrupt vector for receiver interrupts.
;Use MSDOS function calls wherever possible and BIOS routines
;for communications, and graphic control.

MY_CODE SEGMENT PARA    PUBLIC 'CODE'
        ASSUME  CS:MY_CODE,DS:MY_DATA,ES:MY_DATA,SS:MY_STACK

TELE_COMM       PROC    FAR
START:
        PUSH    DS              ;Save segment
        XOR     AX,AX           ;Save returning offset
        PUSH    AX
        MOV     AX,MY_DATA      ;Set up segment registers
        MOV     DS,AX

        MOV     ES,AX
        CALL    INIT            ;Initialize communications port

MAIN_LOOP:
        CALL    RECEIVE         ;See if any characters in RX buffer
        CALL    KEYBOARD        ;See if there are any characters
                                ;to transmit.
        CALL    TX_RS232        ;Se if we should send any characters
        CMP     EXIT_FLAG,01H   ;Leave the program?
        JZ      EXIT            ;If = 01 then leave
        CALL    PRINTER_OUT_1   ;Send to the printer routine
        JMP     MAIN_LOOP       ;Go check main loop.

EXIT:
;********
;disable RS232 interrupts. Reset bit 3 of the modem status register
        XOR     AX,AX
        MOV     DX,MODEM_CONTROL
        OUT     DX,AL
;********
;disable RS232 interrupts via the UART interrupt enable register
;
        MOV     DX,INTERRUPT_ENABLE
        OUT     DX,AL

;*******
;Mask communications interrupts by writing to the 8259's interrupt
;mask register.
        MOV     AL,10101000B            ;Leave disk, timer, keyboard,
                                        ;and graphics bd., interrupts
                                        ;enabled.
        MOV     DX,INT_CONTROL          ;
        OUT     DX,AL

;Establish old RS232 interrupt vector before leaving.
        PUSH    DS              ;Save vector
        MOV     DX,WORD PTR OLD_VECTOR ;Get the old offset
        MOV     AX,WORD PTR OLD_VECTOR+2 ;Set up new vector
        MOV     DS,AX           ;Get old vector segment
        MOV     AL,0CH          ;COMM. interrupts
        MOV     AH,F_SET_INT_VECTOR
        INT     21H
        POP     DS              ;Get old vector
;************

        RET
TELE_COMM       ENDP
;*******************************************************************
;This procedure will transmit characters from the TX_BUFFER.
```

Listing 10-2 continued

```
;
TX_RS232        PROC    NEAR
        CMP     TX_CHARS,00H            ;If there are no characters
                                        ;to transmit, exit this
                                        ;procedure.
        JZ      EXIT_TX_RS232           ;
        MOV     DX,SERIAL_PORT          ;Get port number
        @RS_STATUS                      ;Get the RS232 status
        TEST    AH,00100000B            ;Is the THR empty?
        JZ      EXIT_TX_RS232           ;If not, then leave this
                                        ;procedure
        TEST    SYSTEM_STATUS,00111100B ;See if we are DC3'd by the
                                        ;keyboard, printer, or receive
                                        ;buffer full.
        JNZ     EXIT_TX_RS232           ;If any are set, then exit-
                                        ;do not send.
        MOV     SI,TX_BUFFER_OUT        ;Get the pointer into the buffer
        MOV     AL,BYTE PTR TX_BUFFER[SI] ;Get the character from
                                        ;the buffer.
        CMP     AL,DC3                  ;Are we sending a DC3?
        JNZ     G_S_0                   ;No, then check for DC1
        OR      SYSTEM_STATUS,00100000B ;Set bit 5 to indicate we sent
                                        ;a DC3 from the keyboard.
        JMP     SHORT   GO_SEND         ;Send the DC3
G_S_0:
        CMP     AL,1AH                  ;End of file garbage during
                                        ;S-record downloads.
        JNZ     G_S_0_1
        CALL    TX_CLR
        JMP     EXIT_TX_RS232

G_S_0_1:

        CMP     AL,DC1                  ;A DC1?
        JNZ     GO_SEND
        AND     SYSTEM_STATUS,11011111B ;Reset bit 5 to clear kybd DC3

GO_SEND:
        PUSH    AX                      ;Save character
        MOV     DX,THR                  ;Transmitter holding register
                                        ;of UART
        OUT     DX,AL                   ;Send the character
        POP     DX                      ;Retrieve character
        PUSH    DX
ECHO_TEST:
        TEST    SYSTEM_STATUS,01000000B ;If bit 6 is reset, then do not
                                        ;echo character.
        JZ      NO_ECHO_3               ;IFZ then skip display.
        CMP     DL,11H
        JZ      NO_ECHO_3
        CMP     DL,13H
        JZ      NO_ECHO_3               ;Do not display pacing chars.
NO_E_T:
        CMP     DL,0DH                  ;If char = CR and the
                                        ;local echo option is on,
                                        ;do a linefeed.
        JNZ     NO_E_T_2                ;If not, then skip.
        MOV     DL,0AH
        @CHARDSP
        MOV     DL,0DH
NO_E_T_2:
        CMP     DL,0AH                  ;If char = LF and local echo
                                        ;is on, then skip.
        JZ      NO_ECHO_3

        @CHARDSP

NO_ECHO_3:
        POP     AX                      ;Retrieve character
        INC     TX_BUFFER_OUT           ;Inc. the pointer
        AND     TX_BUFFER_OUT,00FFH     ;Keep < = 256 bytes
        DEC     TX_CHARS                ;Decrement number of chars.
                                        ;in the buffer.
EXIT_TX_RS232:
        RET
TX_CLR:
        MOV     TX_CHARS,0              ;Zero pointers to buffer
        MOV     TX_BUFFER_IN,0
        MOV     TX_BUFFER_OUT,0
        RET

TX_RS232        ENDP

;*************************************************************
;This procedure will fetch a character from the keyboard if one is
;ready. If one is not, the routine will return to the main program.
KEYBOARD        PROC    NEAR
        TEST    SYSTEM_STATUS_S,11111111B ;If non-zero then go read
                                ;from disk file.
        JZ      KEY_2           ;If not set, then read keys.
        CALL    START_TX_FILE   ;Read from disk.
        JMP     DONE_KEYBOARD   ;DONE
KEY_2:
        MOV     DL,0FFH         ;Select keyboard read.
        @CON_IO                 ;See if a character is ready
        JNZ     KEY_3
        JMP     DONE_KEYBOARD   ;If ZF = 1 then no character
KEY_3:
        CMP     AL,00H          ;If 00 then EXTENDED character
        JNE     K_1

CHECK_EXTEND:
        @CON_IO         ;Get the keyboard extended scan code
```

Listing 10-2 continued

```
        CMP     AL,48D          ;alt-b = read from disk
        JNZ     E_K_2           ;go check next code
        CALL    START_TX_FILE
        JMP     DONE_KEYBOARD

E_K_2:
        CMP     AL,32D          ;If ALT-D then enable/disable disk spool.
        JNE     E_K_3           ;Continue if not.
        CALL    TURN_DISK_ON_OFF        ;Go open file.
        JMP     SHORT DONE_KEYBOARD ;
E_K_3:
        CMP     AL,44D          ;If alt - z then exit program.
        JNE     E_K_4
        MOV     EXIT_FLAG,01    ;Set flag to leave program
        JMP     SHORT DONE_KEYBOARD
E_K_4:
        CMP     AL,35D          ;Help command?
        JNE     DONE_KEYBOARD   ;If not then leave
        CALL    HELP            ;Otherwise display commands
        JMP     SHORT DONE_KEYBOARD


K_1:    CMP     AL,DC1          ;Send a DC1?
        JNZ     K_0             ;Go if not.
        TEST    SYSTEM_STATUS,00100000B ;Are we DC3'd?
        JZ      K_0             ;If not then go to next portion of code.
        MOV     DX,THR          ;Transmit the control
        OUT     DX,AL           ;Send it.
        AND     SYSTEM_STATUS,11011111B ;Reset the keyboard DC3 flag bit.
        JMP     DONE_KEYBOARD   ;Do not put into buffer.
K_0:
        CMP     AL,DC2          ;Turn printer on?
        JNZ     K_2             ;If not then continue.
        CALL    PRINTER_ON      ;Go turn the printer on.
                                ;Send this character to the comm buffer.
        JMP     DONE_KEYBOARD   ;Do not store character in TX buffer
K_2:
        CMP     AL,DC4          ;See if DC4 = turn printer off
        JNZ     K_4
        CALL    PRINTER_OFF ;Go turn the printer off.
        JMP     DONE_KEYBOARD ;Do not store in buffer

K_4:

STUFF_BUFFER:
        MOV     SI,TX_BUFFER_IN ;Put the characxter in AL into
                                ;the transmit buffer.
        MOV     TX_BUFFER[SI],AL ;
        INC     TX_BUFFER_IN    ;Increment the pointer
        AND     TX_BUFFER_IN,00FFH ;Keep the buffer circular
        INC     TX_CHARS        ;and number of characters in the buffer.
        MOV     DL,AL           ;See if we need to put the character
                                ;in the print buffer or disk buffer
        TEST    SYSTEM_STATUS,01000000B ;If no echo, then skip
        JZ      DONE_KEYBOARD   ;Skip if no echo.
        CALL    PRINTER_IN_1
        CALL    DISK_IN_1
DONE_KEYBOARD:
        RET
HELP:
        @SCROLL 06,00,18H,4FH,00,00,07
        @VDLINE HMES1                   ;Display help messages-ALT B
        @VDLINE HMES2                   ;ALT-D
        @VDLINE HMES3                   ;ALT-H
        @VDLINE HMES4                   ;ALT-Z
        @VDLINE HMES5                   ;DC1,DC2,DC3,DC4
        RET

KEYBOARD        ENDP

;**********************************************************************
;This routine will set the printer status to on for printer spooling
;
PRINTER_ON      PROC    NEAR
        PUSH    AX                      ;Save character
        OR      SYSTEM_STATUS,00000001B ;Set b0 to 1 to indicate
                                        ;the printer is on.
        @CRLF
        @VDLINE MESSAGE18               ;Display printer on message
        POP     AX
        RET
PRINTER_ON      ENDP
;**********************************************************************
PRINTER_OFF     PROC    NEAR
        PUSH    AX                      ;Save character
        AND     SYSTEM_STATUS,0FEH      ;Reset system status
        @CRLF
        @VDLINE MESSAGE19               ;Show printer is off

        POP     AX                      ;Restore character
        RET                             ;Bit 0 to indicate the printer
                                        ;spooling is off.
PRINTER_OFF     ENDP
;**********************************************************************
;This procedure will toggle the disk spooling feature.
;If the procedure is off, it will ask for the file name, and
;open a buffer for character routing to disk.
;If the feature has already been selected, the procedure will
;turn the spooling feature off, and close the file.
;Assumes a file handle to be stored in memory location
;'FILE_HANDLE'.
;
```

Listing 10-2 continued

```
TURN_DISK_ON_OFF        PROC    NEAR
        TEST    SYSTEM_STATUS,00000010B ;If b1 = 1 then disk is on
                                        ;so turn it off and close file.
        JZ      TDISK_1                 ;Zero? then turn it on.
        JMP     TURN_DISK_OFF   ;Go turn the disk off.

TDISK_1:
        MOV     DL,DC3          ;Send a DC3 to the other terminal
                                ;suspend communications for a moment.
        @CRLF                   ;Go to the next line
        @VDLINE MESSAGE17       ;Ask for the filename
        @KBDLINE KBUFFER        ;Get the pathname.
        XOR     BH,BH           ;Zero high order half of BX register.
        MOV     BL,CHARS_TYPED  ;Offset to last char typed
        MOV     KBDBUFFER[BX],00H ;Terminate with a zero
        LEA     DX,KBDBUFFER    ;Point to file name
        MOV     CX,00H          ;Read write access
        MOV     AH,F_CREATE_FILE_2
        INT     21H

        JNC     GOOD_OPENING    ;No carry means file open and
                                ;AX holds the file handle.
        @CRLF
        @VDLINE MESSAGE15       ;Display error message.
        @CRLF
        JMP     DONE_DISK       ;All done
GOOD_OPENING:
        MOV     FILE_HANDLE,AX  ;Save the file handle
        @SET_DTA        DISK_BUFFER ;Set the DTA
        OR      SYSTEM_STATUS,00000010B ;Set status to disk on.
        @CRLF
        @VDLINE MESSAGE20       ;Show the file is open
        JMP     DONE_DISK
TURN_DISK_OFF:
        AND     SYSTEM_STATUS,11111101B ;Reset b1 to indicate disk
                                        ;spooling is off.
        @VDLINE MESSAGE21               ;Display file closed message
        LEA     DX,DISK_BUFFER          ;Write remaining buffer
                                        ;to disk.
        MOV     CX,DISK_IN              ;Number of characters to write.
        MOV     BX,FILE_HANDLE          ;Get the file handle
        MOV     AH,F_WRITE_FILE_2       ;MSDOS function code
        INT     21H                     ;MSDOS function interrupt.

        @CLOSE_FILE_2 FILE_HANDLE       ;Close the file
        JNC     DONE_DISK               ;Go if no errors
        @VDLINE MESSAGE15               ;Display error message.
DONE_DISK:
        LEA     SI,KBDBUFFER    ;Point to the keyboard buffer
        MOV     CX,32           ;Clear the pathname form the buffer
```

```
CLEAR_KBUFFER:
        MOV     BYTE PTR [SI],0 ;Clear the buffer
        INC     SI              ;Point to next location to clear
        LOOP    CLEAR_KBUFFER
        RET
TURN_DISK_ON_OFF        ENDP
;**********************************************************************
;*********************************************************************
;This procedure is the recive handler. It will process characters
;received from RS-232 interrupts which were placed in the RS-BUFFER.
;The procedure is executed at task time and not at interrupt time.
;
RECEIVE PROC    NEAR
        CMP     RS_CHARS,00     ;If RS_CHARS = 0 then return, there
                                ;are no characters in the buffer
        JNE     GET_BUFF_CHAR   ;Exit if zero
        JMP     DONE_RECEIVE

GET_BUFF_CHAR:
        MOV     SI,RS_OUT_POINT ;Point to the next character to
                                ;remove
        MOV     DL,RS_BUFFER[SI]        ;Get the character

        AND     DL,01111111B    ;Reset bit 8 - display only ASCII
        CMP     DL,1AH                  ;Is it an ASCII sub?
        JNZ     G_B_2                   ;If not then continue
        MOV     DL,'?'                  ;Use ? character
                                        ;to signify an error
        JMP     SHORT   DSP_CHAR        ;Go display the character
G_B_2:
        CMP     DL,20H                  ;Control or character?
        JAE     DSP_CHAR                ;Character then go display it.
        CMP     DL,0DH                  ;Carriage return?
        JZ      CRLF                    ;Do a carriage return linefeed
        CMP     DL,07H                  ;BELL?
        JZ      DSP_CHAR                ;Go display
        CMP     DL,08H                  ;BACKSPACE?
        JZ      DSP_CHAR
        CMP     DL,09H                  ;TAB?
        JZ      DSP_CHAR
        CMP     DL,0BH                  ;VERTICAL TAB?
        JZ      DSP_CHAR                ;If so, display the control
        CMP     DL,11H                  ;DC1?
        JZ      TX_TOGGLE_DC1           ;Go toggle the TX control
        CMP     DL,13H                  ;DC3?
        JZ      TX_TOGGLE_DC3           ;Toggle the control
        JMP     G_B_3_1                 ;Ignore other controls

DSP_CHAR:
        @CHARDSP                        ;Display the character in DL

G_B_3:
```

Listing 10-2 continued

```
        CALL    PRINTER_IN_1            ;See if we should place the
                                        ;character into the printer buffer
        CALL    DISK_IN_1               ;See if we should place the
                                        ;character in the disk buffer.
G_B_3_1:

        INC     RS_OUT_POINT            ;Increment pointer
        AND     RS_OUT_POINT,07FFH      ;Keep circular
        DEC     RS_CHARS                ;Decrement number of characters
                                        ;in the buffer.
        JMP     SHORT DONE_RECEIVE      ;And exit
CRLF:
        TEST    SYSTEM_STATUS,80H       ;If bit 7 is set then display
                                        ;new line.
        JZ      CR_ONLY                 ;If not, then display only the CR
        MOV     DL,0AH                  ;Do a LINEFEED (0A HEX)
        @CHARDSP                        ;Display it.
        CALL    PRINTER_IN_1            ;See if we should place the
                                        ;character into the printer buffer
        CALL    DISK_IN_1               ;See if we should place the
CR_ONLY:
        MOV     DL,0DH                  ;Now the carriage return
        @CHARDSP

        JMP     SHORT   G_B_3           ;Increment pointers
TX_TOGGLE_DC1:
        AND     SYSTEM_STATUS,11110111B ;Reset bit 3 to reflect
                                        ;receiving a DC1
        JMP     SHORT   G_B_3_1
TX_TOGGLE_DC3:
        OR      SYSTEM_STATUS,00001000B ;Set bit 3 to reflect receiving
                                        ;DC3
        JMP     SHORT   G_B_3_1
DONE_RECEIVE:
        RET
RECEIVE ENDP

;**********************************************************************
;This procedure will place the character in DL into the printer buffer
;if the printer is enabled.
;
PRINTER_IN_1    PROC    NEAR
        PUSH    SI                      ;Save SI
        PUSH    DX                      ;Save DX
        PUSH    AX
        TEST    SYSTEM_STATUS,01H       ;See if the printer is on
        JZ      EXIT_PRINTER_IN         ;If not, then exit this procedure.
        MOV     SI,PRINTER_IN           ;Get the pointer
        MOV     PRINTER_BUFFER[SI],DL   ;Put character in the printer buffer
        INC     PRINTER_IN              ;Inc pointer
        AND     PRINTER_IN,00FFH        ;Keep circular
```

```
       INC     PRINTER_CHARS           ;Active chars in buffer incremented.
       CMP     PRINTER_CHARS,200       ;If chars > 200 then halt until
                                       ;buffer clears
       JBE     EXIT_PRINTER_IN         ;If not see exit the routine
       MOV     AL,DC3                  ;Send a DC3 to halt transmissions
       MOV     DX,THR                  ;Send the control
       OUT     DX,AL
       OR      SYSTEM_STATUS,04H       ;Set bit 2, to show printer DC3'd

EXIT_PRINTER_IN:
       POP     AX
       POP     DX                      ;Retrieve registers
       POP     SI
       RET
PRINTER_IN_1   ENDP
;****************************************************************
;This procedure will place the character in DL into the disk buffer
;if the buffer is opened. If not, the routine is terminated.
;The routine will perform a write to disk if the character count
;in the buffer equals one 512 byte sector.
;If there is an error, the routine will report such, and close
;the file.
;
DISK_IN_1      PROC    NEAR
       PUSH    SI                      ;Save registers
       PUSH    AX
       PUSH    DX
       PUSH    BX
       MOV     BX,THR                  ;Use as port address
                                       ;to send character to when
                                       ;pacing is required.
       TEST    SYSTEM_STATUS,02H       ;See if bit 2 is set
                                       ;if non-zero result, then
                                       ;disk spooling is on
       JZ      EXIT_DISK_1             ;Leave routine

       CMP     DL,20H                  ;If less than 20H, do not
                                       ;send to disk buffer
       JAE     GOTO_DISK               ;If > = to 20H then skip
                                       ;control character check.
CHECK_CONTROLS:
       CMP     DL,0AH                  ;
       JZ      GOTO_DISK               ;Send to disk
       CMP     DL,0DH                  ;CR is ok to send to disk
       JZ      GOTO_DISK               ;

       CMP     DL,06H                  ;Backspace is ok.
       JZ      GOTO_DISK               ;
       JMP     EXIT_DISK_1             ;Any other controls are tossed.

GOTO_DISK:
       CMP     DISK_IN,SECTOR-10       ;Start pacing if near end of buffer
```

Listing 10-2 continued

```
        JB      CONT_DISK_IN            ;Keep going otherwise
        XCHG    DX,BX                   ;Transmitter port address to
                                        ;DX; and save character in BL

        MOV     AL,DC3                  ;Send control character
        OUT     DX,AL                   ;Control sent to other terminal
        INC     DISK_STATUS             ;Non-zero byte means DC3'd
        XCHG    DX,BX                   ;Retrieve character
CONT_DISK_IN:
        MOV     SI,DISK_IN              ;Get pointer into buffer
        MOV     DISK_BUFFER[SI],DL      ;Save character in buffer
        INC     DISK_IN                 ;Increment the number of characters
                                        ;in the DTA
        CMP     DISK_IN,SECTOR          ;If the characters in buffer are
                                        ;= 512 then it is time to
                                        ;write a sector to disk.
        JZ      DISK_WRITE              ;Leave routine if the number is below
        JMP     SHORT   EXIT_DISK_1
DISK_WRITE:
        PUSH    DX                      ;Save registers
        PUSH    CX
        PUSH    AX
        PUSH    BX
        LEA     DX,DISK_BUFFER          ;Set up registers for disk write.
        MOV     CX,SECTOR               ;512 BYTES TO WRITE
        MOV     AH,F_WRITE_FILE_2       ;MSDOS function code
        MOV     BX,FILE_HANDLE          ;File handle of open file
        INT     21H                     ;MSDOS function interrupt
        POP     BX                      ;Retrieve registers
        POP     AX
        POP     CX
        POP     DX
        JNC     WRITE_DONE              ;The write has been performed
DISK_ERROR:
        @VDLINE MESSAGE15               ;Display disk write error
        @CLOSE_FILE_2 FILE_HANDLE       ;Close the file.
        AND     SYSTEM_STATUS,0FDH      ;Reset bit 1 to show disk
                                        ;access is off.
WRITE_DONE:
        MOV     DISK_IN,0               ;Reset pointer
EXIT_DISK_1:
        CMP     DISK_STATUS,00H         ;If zero then leave
        JZ      DISK_EXIT_2
SEND_DC1:
        XCHG    DX,BX                   ;Get the port address
        MOV     AL,DC1                  ;for the transmitter
        OUT     DX,AL                   ;Send the control char.
        XCHG    DX,BX                   ;Restore character
        MOV     DISK_STATUS,00H         ;Clear memory
DISK_EXIT_2:
        POP     BX
        POP     DX
        POP     AX
        POP     SI                          ;Restore registers
        RET
DISK_IN_1       ENDP
;**********************************************************************
;This routine will remove a character from the printer buffer
;and send it to the printer if the printer option is enabled.
;This routine uses BIOS to send a character to the printer.
;The character to print must be in AL.
;
PRINTER_OUT_1   PROC    NEAR
        PUSH    DX
        PUSH    SI              ;Save registers
        PUSH    AX
        TEST    SYSTEM_STATUS,00000001B ;If bit 0 is a 1 then printer is on.
        JZ      EXIT_PRINTER_OUT        ;If b0 = 0 then off, so leave
        @PRINTER_STATUS                 ;See if the printer is on line
                                        ;and ready.
        TEST    AH,10000000B            ;If the printer is busy then
                                        ;skip this routine for now.
        JZ      EXIT_PRINTER_OUT        ;bit 7 = 0 = busy
                                        ;        1 = not busy
OK_CONTINUE:

        CMP     PRINTER_CHARS,00H       ;Any in buffer?
        JZ      EXIT_PRINTER_OUT        ;If none in buffer, exit.
        MOV     SI,PRINTER_OUT          ;Get out pointer
        MOV     AL,PRINTER_BUFFER[SI]   ;Get character
        CMP     AL,0DH                  ;Carriage return?
        JNZ     CONT_PRT                ;No? Then continue.
        @PRINTER_OUT_B                  ;
        MOV     CX,7000H                ;Wait value if busy.
G_STAT:
        @PRINTER_STATUS                 ;Wait till printer is ready.
        TEST    AH,80H                  ;If not ready, then loop.
        JNZ     CONT_PRT_0              ;Ok, do next character
        LOOP    G_STAT                  ;WAIT TILL CX = 0, AND TRY ANYWAY
CONT_PRT_0: MOV AL,0AH                  ;Now do the linefeed

CONT_PRT: @PRINTER_OUT_B                ;BIOS call to send printer
                                        ;a character in AL.
        INC     PRINTER_OUT             ;Inc pointer to next character
        AND     PRINTER_OUT,00FFH       ;Keep circular
        DEC     PRINTER_CHARS           ;Dec number of characters in buff.
        CMP     PRINTER_CHARS,100       ;If less than 100 characters
                                        ;go see if we are DC3'd
        JB      CHK_P_DC3
EXIT_PRINTER_OUT:
        POP     AX
        POP     SI                      ;Restore registers
        POP     DX
```

Listing 10-2 continued

```
        RET

CHK_P_DC3:
        TEST    SYSTEM_STATUS,00000100B ;See if we were DC3'd
        JZ      EXIT_PRINTER_OUT        ;No, then exit routine.
        AND     SYSTEM_STATUS,11111011B ;Reset bit 2 clearing DC3
        MOV     AL,DC1                  ;Send a DC1
        MOV     DX,THR                  ;
        OUT     DX,AL
        JMP     EXIT_PRINTER_OUT        ;Exit this routine.
PRINTER_OUT_1   ENDP

;*********************************************************************
;This procedure is the RS232 interrupt handler.
;If the interrupt is due to a received character, the character is
;read from the RS232 and placed into a circular buffer which
;will process the received characters at task time, not at interrupt time.
;gThis procedure assumes that a buffer is established in the
;current data segment.
;
RS232_INT       PROC    FAR
RS232_INT_1:
        PUSH    SI
        PUSH    AX
        PUSH    DX
        PUSH    BX
        PUSH    ES
        PUSH    DS
        ASSUME  DS:MY_DATA,ES:MY_DATA
GET_CHAR:
        MOV     AX,MY_DATA
        MOV     DS,AX
        MOV     ES,AX
        MOV     SI,RS_IN_POINT  ;Get pointer into the buffer
GET_RS_CHAR:

        MOV     DX,LINE_STATUS  ;Check for any errors or a
                                ;break condition.
        IN      AL,DX           ;Get the status.
        TEST    AL,00011110B    ;b4 = break
                                ;b3 = framing error
                                ;b2 = parity error
                                ;b1 = overrun error
        JNZ     ERROR           ;If non-zero then there is an
                                ;error.
        TEST    AL,01H          ;If bit 0 = 1 then rcv. char. interrupt
                                ;has occurred. If b0 = 0 then skip routine
        JZ      EXIT_INT_1      ;

        MOV     DX,RHR          ;Receiver port
        IN      AL,DX           ;Get the character
        MOV     RS_BUFFER[SI],AL ;Save character in buffer
        JMP     SHORT   POINT_ADJ ;Leave the interrupt routine
ERROR:
        MOV     DX,RHR          ;Dummy read to reset the interrupt
        IN      AL,DX
        MOV     AL,1AH          ;Code for ASCII SUB (Substitute)
        MOV     RS_BUFFER[SI],AL ;Save it in buffer
POINT_ADJ:
        INC     RS_IN_POINT ;Increment pointer
        AND     RS_IN_POINT,07FFH ;Keep circular
        INC     RS_CHARS        ;Increment number of characters
                                ;in the buffer
EXIT_INT:
        MOV     DX,INTERRUPT_ID ;Get any other pending interrupts
        IN      AL,DX           ;AND TOSS THEM!
EXIT_INT_1:
        MOV     AL,20H          ;Send an end of interrupt to the 8259
        OUT     INT_COMMAND,AL
        POP     DS
        POP     ES
        POP     BX
        POP     DX              ;Restore registers
        POP     AX
        POP     SI
        IRET                    ;Return from the interrupt
        @CRLF
        @WAITKEY                        ;Get user response
        MOV     MIN,31H                 ;Minimum value is '1'
        MOV     MAX,39H                 ;Maximum value allowed = '9'
        CALL    MINMAX                  ;Check bounds
        JNC     SET_BAUD                ;Go to the next prompt if in bounds
        JMP     INIT_1                  ;else go ask again.
;**** Response was in bounds, so proceed.
SET_BAUD: AND   AL,0FH                  ;Strip ASCII character to binary
        DEC     AL                      ;Adjust for desired bit pattern
        MOV     CL,3                    ;Rotate b0-b2 to b7-b5
        ROR     AL,CL                   ;
        AND     CONFIGURATION,00011111B ;Reset b0-b5 of configuration byte
        OR      CONFIGURATION,AL        ;OR in bits.
GET_PARITY:
        CALL    CLEAR_SCREEN            ;Clear screen and ask for
                                        ;Parity.
        @CURSET 0,0,0                   ;Set cursor top of screen
        @VDLINE MESSAGE6                ;Ask Parity?
        @CRLF
        @WAITKEY                        ;Get response
        AND     AL,11011111B            ;Reset bit 5, convert to
                                        ;uppercase.
        CMP     AL,'Y'                  ;If yes, then continue
        JNE     SET_NONE                ;If not go to the next
                                        ;prompt.
EVEN_ODD:
```

Listing 10-2 continued

```
RS232_INT       ENDP

;**************************************************************


;********************************************************************
;This is the initialization module which will prompt the user for the
;various parameters required for communications.
;This routine expects a dedicated area of RAM to be reserved for the
;storage: CONFIGURATION.
;
INIT    PROC    NEAR
INIT_1:
        CALL    CLEAR_SCREEN              ;Clear the screen

        @CURSET 0,0,0                     ;Set cursor to page 0,
                                          ;row 0, column 0
GET_BAUD:
        @VDLINE MESSAGE1                  ;Ask for baud rate
        @CRLF
        @VDLINE MESSAGE2                  ;Show possible rates.
        @CRLF
        @VDLINE MESSAGE3
        @CRLF
        @VDLINE MESSAGE4
        @CRLF
        @VDLINE MESSAGE5                  ;
        CALL    CLEAR_SCREEN              ;Clear the screen
        @CURSET 0,0,0                     ;Position the cursor
        @VDLINE MESSAGE7                  ;Ask even or odd parity?
        @CRLF
        @WAITKEY                          ;Get response
        AND     AL,0FH                    ;Mask to binary
        CMP     AL,1                      ;Even?
        JE      EVEN_P                    ;Then set it
        CMP     AL,2                      ;Odd?
        JNE     EVEN_ODD                  ;If neither 1 or 2 then
                                          ;reask the question.
SET_ODD:
        AND     CONFIGURATION,11101111B ;Reset bit 4
        OR      CONFIGURATION,00001000B ;And set bit 3
        JMP     SHORT GET_STOPS           ;Go to the next prompt
EVEN_P: OR      CONFIGURATION,00011000B ;Set both bits 4 and 3
        JMP     SHORT   GET_STOPS         ;Go to next prompt
SET_NONE: AND   CONFIGURATION,11100111B ;Reset both bits 4 and 3

GET_STOPS:                                ;How many stop bits?
        CALL    CLEAR_SCREEN              ;
        @CURSET 0,0,0                     ;Position cursor
        @VDLINE MESSAGE8                  ;Ask how many
        @CRLF
        @VDLINE MESSAGE9                ;Show options
        @CRLF
        @WAITKEY                        ;Get response
        AND     AL,0FH                  ;Strip to binary
        DEC     AL                      ;Was the response a '1'?
        JZ      ONE_STOP                ;Then setup 1 stop bit
        DEC     AL                      ;iF IT WAS NOT A '2' THEN
                                        ;REASK THE QUESTION.
        JNE     GET_STOPS
TWO_STOPS:
        OR      CONFIGURATION,00000100B ;Set bit 2 for 2 stop bits
        JMP     SHORT   WORD_LENGTH    ;Get word length
ONE_STOP:
        AND     CONFIGURATION,11111011B ;Reset b2 for 1 stop bit
WORD_LENGTH:
        CALL    CLEAR_SCREEN            ;Clear the screen
        @CURSET 0,0,0                   ;Set the cursor
        @VDLINE MESSAGE10               ;Ask word length
        @CRLF
        @VDLINE MESSAGE11               ;Show options
        @CRLF
        @VDLINE MESSAGE12
        @CRLF
        @WAITKEY                        ;Get user response
        CMP     AL,'1'                  ;7 bits?
        JZ      SEVEN_UP                ;
        CMP     AL,'2'                  ;If not 2 then reask the
                                        ;question.
        JNE     WORD_LENGTH
EIGHT_BITS:
        OR      CONFIGURATION,00000011B ;Set both bits 0 and 1
        JMP     SHORT   TO_ECHO_OR_NOT_TO ;Next question.
SEVEN_UP:
        AND     CONFIGURATION,11111110B ;Reset bit 0
        OR      CONFIGURATION,00000010B ;Set bit 1
TO_ECHO_OR_NOT_TO:
        CALL    CLEAR_SCREEN            ;Ask the next question.
        @CURSET 0,0,0                   ;Set the cursor position
        @VDLINE MESSAGE30               ;Ask if the user wants
                                        ;the characters typed locally
                                        ;echoed to the screen. Used in
                                        ;half duplex communications.
        @CRLF                           ;Next line
        @WAITKEY                        ;Wait for a Yes or No answer.
        AND     AL,11011111B            ;Reset bit 5, convert to upper case.
        CMP     AL,'Y'                  ;Yes? Then set bit 6 of
                                        ;system_status to reflect such.
        JNZ     NO_ECHO                 ;If not, then clear the bit.
        OR      SYSTEM_STATUS,01000000B ;Set bit 6 in response to Yes.
        JMP     SHORT   NEW_LINE_Q      ;Ask the next question.
```

Listing 10-2 continued

```
NO_ECHO:
        AND     SYSTEM_STATUS,10111111B ;Reset bit 6, no local echo.
NEW_LINE_Q:
        CALL    CLEAR_SCREEN            ;Clear the screen and set the cursor.
        @CURSET 0,0,0                   ;
        @VDLINE MESSAGE31               ;Ask the question, do a CRLF
                                        ;combination on receipt of a CR?
        @CRLF                           ;
        @WAITKEY                        ;Wait for a Y or N
        AND     AL,11011111B            ;Convert to upper case
        CMP     AL,'Y'                  ;Yes? then set bit 7. Else clear bit
                                        ;7 in SYSTEM_STATUS.
        JNZ     NO_NEW_LINE             ;If not then reset the bit.
        OR      SYSTEM_STATUS,10000000B ;New line function on.
        JMP     SHORT   SET_CONFIGURATION
NO_NEW_LINE:
        AND     SYSTEM_STATUS,01111111B ;Reset the bit.

SET_CONFIGURATION:
        @RS232_INIT CONFIGURATION       ;Set the RS-232 port

;**************
;Now set a new RS-232 interrupt vector. Save the old.
        PUSH    ES                      ;Save extra segment value
        PUSH    BX                      ;And BX
        @READ_VECTOR 0CH                ;Get the vector for
                                        ;RS232 interrupts
        MOV     WORD PTR OLD_VECTOR,BX  ;Save old vector
        MOV     WORD PTR OLD_VECTOR[2],ES ;Save Old segment addr.
        POP     BX                      ;Retrieve BX and ES
        POP     ES                      ;
        LEA     DX,RS232_INT_1          ;Get new offset
        PUSH    DS                      ;Save data segment
        MOV     AX,MY_CODE              ;Segment of int. routine
        MOV     DS,AX                   ;Seg of int. routine
        MOV     AL,0CH                  ;Interrupt Type
        MOV     AH,F_SET_INT_VECTOR     ;Function code
        INT     21H                     ;Go set new vector
        POP     DS                      ;Retrieve DS
        CLI                             ;disable interrupts

;**********
;Now set RTS and DTR true, (logic 0).
;Bit 3 must be a '1' to enable interrupts to the 8088.

        MOV     AL,00001011B            ;Set RTS, DTR, true and
                                        ;enable interrupts (bit 3)
        MOV     DX,MODEM_CONTROL        ;Set up the modem control register
        OUT     DX,AL

;***********
```

```
;Set up the serial board for receiver interrupts.
;First clear any pending interrupts.

        MOV     DX,LINE_STATUS          ;Clear the line status int.
                                        ;if any, by reading the
                                        ;register (dummy read)
        IN      AL,DX                   ;Means nothing.

        MOV     AL,00000001B            ;Enable interrupts on
                                        ;comm. board when received
                                        ;character is ready.
        MOV     DX,INTERRUPT_ENABLE
        OUT     DX,AL


;*******
;Now set the 8259 to enable the desired interrupt types.
;
        MOV     AL,10101100B            ;Enable disk, comm,
                                        ;keyboard, and timer interrupts.
        OUT     INT_CONTROL,AL          ;Send to 8259

        STI                             ;Enable 8088 interrupts


        @CRLF                           ;Go to the next line
        @VDLINE MESSAGE13               ;Show message that port was set
        @CRLF
        CALL    DELAY                   ;Leave message on the screen
                                        ;for awhile
                                        ;and erase and show copyright
        CALL    DELAY
        CALL    DELAY
        CALL    CLEAR_SCREEN
        @CURSET 0,0,0
        @VDLINE MESSAGE14
        @CRLF
        RET                             ;All done
INIT    ENDP
;*****************************************************************
;Check bounds on the character in AL. If in bounds, reset carry,
;if out of bounds, set the carry and return.
;
MINMAX  PROC    NEAR
        CMP     AL,MIN          ;If below, set the carry
        JB      SET_CARRY
        CMP     AL,MAX          ;If above, set the carry
        JA      SET_CARRY
        CLC                     ;Else clear the carry
        JMP     SHORT DONE_MINMAX
SET_CARRY:
        STC
```

Listing 10-2 continued

```
DONE_MINMAX: RET
MINMAX  ENDP
;**************************************************************
;Procedure to clear the screen.
CLEAR_SCREEN    PROC    NEAR
        PUSH    AX                          ;Save registers
        PUSH    BX
        PUSH    CX
        PUSH    DX
        @SCROLL 06,00,18H,4FH,00,00,07  ;Clear the screen
        POP     DX                          ;Restore registers
        POP     CX
        POP     BX
        POP     AX
        RET
CLEAR_SCREEN    ENDP
;*************************************************************
;This routine will provide a delay for a fixed duration.
;all registers used are saved.
DELAY   PROC    NEAR
        PUSH    CX      ;Save the registers
        MOV     CX,7000H
DELAY_1: LOOP   DELAY_1
        MOV     CX,3000H
DELAY_2:        LOOP DELAY_2
        POP     CX
        RET
DELAY   ENDP
;******************************************************************
MY_CODE ENDS
        END     START
```

Bibliography

1. *The IBM Macro Assembler Manual by Microsoft*, #6024002. Boca Raton, FL: IBM, 1981.
2. *The IBM Technical Reference Manual* #6025005, Boca Raton, FL: IBM, 1983.
3. *The IBM 2.10 DOS Technical Reference Manual by Microsoft*, #6024125. Boca Raton, FL: IBM, 1983.
4. *Intel ASM86 Reference Manual*, Intel Corporation, Order Number 121703-002.
5. *Intel iAPX 88 Book*, Intel Corporation, Order Number 210200-002.
6. *Intel iAPX 86,88 User's Manual*, Intel Corporation, Order Number 210201-001.
7. *Intel iAPX 86/88, 186/188 User's Manual—Programmer's Reference*, Intel Corporation, Order Number 210911-001.
8. *Intel Microprocessor and Peripheral Handbook*, Intel Corporation, Order Number 210844-001.
9. Martin, James. *Design and Strategy for Distributed Processing*. Englewood Cliffs, NJ: Prentice Hall Inc., 1981.
10. Martin, James. *Telecommunications and the Computer*. Englewood Cliffs, NJ: Prentice Hall Inc., 1976.

Index

AAA (ASCIIAdjust for Addition) instruction, 90, 328
AAD (ASCII Adjust for Division) instruction, 101, 328
AAM (ASCII Adjust for Multiplication) instruction, 99, 328
AAS (ASCII Adjust for Subtraction) instruction, 92, 328
absolute disk access, 229, 243
ACK (acknowledgement), 284
acoustic coupler, 289
ADC (add with carry) instruction, 88, 329
ADD (addition) instruction, 87, 329
addition
  binary, 4
  binary coded decimal, 9
  hexadecimal, 8
addition instructions 73, 87-91
address bus, 51, 53
address-object transfers, 72, 84
addressing modes, 65-68
  time calculation and, 68
AF (auxiliary carry flag), 61
AH register, 20, 47
AL register, 47
ALE signal, 53
ALU (Arithmetic Logic Unit), 46
ampersand (&), 176, 179
amplitude modulation, 278
analog data transmission, 276-278
AND instruction, 102, 329
AND operator, 167
architecture
  defined, 43
arithmetic instructions, 73, 87-102
arithmetic operators, 167
ASCII (American Standard Code for Information Exchange), 11, 321
  communications and, 268
  conversion to binary system, 202
assembler, xii, 21
  cross, 27
  format for, 136
  use of, 37
ASSUME statement, 127, 136
asterisk (*) operator (multiplication), 167
asymmetric register, 59
asynchronous communications, 282
attribute operators, 162
attributes for labels, 142
auxiliary carry flag, 61
AX register, 47, 50
  POP instruction and, 79

bandwidth, 278
base pointer register, 59
based indexing addressing, 66
baud rate, 285
baud rate divisor registers, 295
Baudot code, 12, 321
BCD (binary coded decimal system), 9
binary coded decimal system, 9
binary system, 2-7
  conversion from ASCII to, 202
  conversion from decimal to, 198
  conversion to decimal, 202
  conversion to hexadecimal, 203
BIOS (Basic Input/Output System), 47, 249-264
  communications functions in, 303
  scroll function in, 233
  video functions in, 258
bit, defined, 2
BIU (Bus Interface Unit), 44
bootstrap, 47
bounds checking, 246
BP (base pointer) register, 59
BPS (bits per second), 285
buffers, data, 309
bus architecture, 51-55
bus controller, 53
byte, defined, 2

calls, function, 182
CALL instruction, 49, 50, 112-114, 329
carry flag, 60
CBW (convert byte to word) instruction, 98, 329
CF (carry flag), 60
characters, encoding of, 11
CLC (clear carry)instruction, 131, 330
CLD instruction, 330
CLI (Clear Interrupt flag) instruction, 132, 330
clusters, defined, 210
CMC (complement carry bit) instruction, 131, 330
CMP (compare) instruction, 94, 330
CMPS (compare string)instructions, 127, 330
code segment, programming and, 192, 233
code segment register, 49
colon suffix, 29
color character attributes, 255
color combinations, 256
color/graphics adapter, 254
COMM.ASM program, 443
comment field in source statement, 31
COMMENT pseudo-op, 148
communications, 267-318
  asynchronous, 282
  data transmission methods and, 270
  computer to computer, 291
  error checking in, 285
      errors in, 300
  parallel, 270
  parameters for, 305
  program for, 292-302, 305-317
  protocols for, 268
  RAM and, 118
  serial, 271
  synchronous, 283
compiled languages, 19
computer languages, types of, 17-21
conditional pseudo-ops, 155-158
conditional transfer instructions, 74, 115-119
CONFIGSY.ASM program, 355
CONFIGSY.LST program, 361
control bus, 51
control signals, 51
copying files, 238-243
CPU (Central Processing Unit), 17
  defined, 43
  memory management and, 46
.CREF pseudo-op, 159, 160
cross assemblers, 27
cross reference file, 38, 40
CS register, 58
CWD (convert word to double word) instruction, 98, 331

DAA (Decimal Adjust for Addition) instruction, 90, 331
DAS (Decimal Adjust for Subtraction) instruction, 93, 331
data buffers, 309
data bus, 51, 53
data pseudo-ops, 136-140
data register, 59
data segment, programming and, 192, 232
data storage, memory and, 47
data transfer instructions, 72, 75-87
  general purpose, 76-82
data transmission methods, 270
  analog, 276
DB pseudo-op, 141
DD pseudo-op, 141
DEC (decrement) instruction, 94, 331
decimal system, 2
  conversion from binary, 202
  conversion to binary, 7, 198
  conversion to hexadecimal, 198
definitions, programming and, 194
DEN signal, 53
destination operand, 71
DF (direction flag), 63
dibit phase encoding, 277
direction flag, 63
directory in MS-DOS, 214
  format of, 215
DIRREAD.ASM program, 426
disk access, absolute, 229
disk buffer, programming and, 232
disk drives, 210
disk I/O, 207-247
  MS-DOS functions and, 207-208

disk sectors, 230
disk transfer address (DTA), 220
diskette, 209
  capacities of, 211
displacement addressing, 67
displaying data, 204
distance attribute, 143, 165
DIV instruction, 99, 331
division instructions, 73, 99-102
DLOAD.ASM program, 447
documentation, comment field and, 31
dollar sign ($), 148
DOSEQU.EQU program, 396
downloading files, 317
DQ pseudo-op, 141
DRAM (Dynamic Random Access Memory), 46
drawing lines, 260
DS operator, 165
DT pseudo-op, 141
dummy arguments, 170
DUP pseudo-op, 141
duplex, 269
DURATION of sound, 263
DW pseudo-op, 141

8086 microprocessor, 53
  instruction set for, 327-345. See also name of instruction
8088 microprocessor
  architecture of, 55
  instruction set for, 71-134, 327-345. See also name of instruction
  interrupts and, 63, 123
  segmentation of, 56
8288 bus controller, 53
EBCDIC code, 12, 321
editor, 23
electrical pacing, 294
ELSE pseudo-op, 157
END pseudo-op, 155
ENDIF pseudo-op, 157
ENDM pseudo-op, 175
ENDP pseudo-op, 139
ENDS pseudo-op, 138
ENDS pseudo-op, 152
EQ operator, 166
EQU (equate) pseudo-op, 145
equal (=) pseudo-op, 145
equates, programming and, 194
erasing lines, 262
error checking in communications, 269, 285-288
error codes for absolute disk I/O, 231
errors in programming, 237
ES operator, 165
  ESC (escape) instruction, 133, 331
EU (Execution Unit), 44
EVEN pseudo-op, 150
exchange (XCHG) instruction, 82, 343
exclamation point (!), 176
exclusive OR (XOR) instruction, 105, 345
EXITM pseudo-op, 175, 178
expressions, 166
extensions, signed, 97
external mask interrupt (INT) instruction, 333
external synchronization, 75, 132-134
EXTRN pseudo-op, 147

FAR procedure, defined, 113
FAST—CPY.ASM program, 421
fields, 28-32
  comment, 31
  label, 29
  op-code, 30
  operand, 31
file
  access to, 221-229
  closing of, 226, 229
  created with assembler, 38-40
  creation, 221, 228
  opening of, 227, 235
file allocation table, 213
file control blocks (FCB), 217
  formats of, 218
  programming and, 233
  specifying, 219
file handles, 226
flag operations, 75, 131-132
flag register, 45, 60
flag register transfers, 72, 86
flow chart
  programming techniques and, 190
  source code and, 34-37
  symbols used in, 33
FOR statement, 119
format for assembler, 136
forward references, 26
framing error, 300
frequency modulation, 277
FREQUENCY of sound, 263
FSK (frequency shift keying), 277, 285, 288
full duplex, 269, 278
full screen editor, 23
function calls, 182

GE operator, 166
GRAPHIC.ASM program, 435
graphics, 254, 256-263
GROUP pseudo-op, 154
GT operator, 166

half duplex, 269, 280
hardware requirements, xii
hexadecimal system, 7, 321
  conversion from binary to, 203
  conversion from decimal to, 198
high level languages, 18
high resolution graphics mode, 257
HIGH operator, 164
HLT (Halt) instruction, 132, 332

IBM PC memory map, 55, 57
IDIV (Integer Divide) instruction, 100, 332
IF (interrupt flag), 62
IF pseudo-ops, 157
immediate addressing, 65
immediate operand, 72
IMUL (Integer Multiply) instruction, 97, 332
IN instruction, 83, 332
INC (increment) instruction, 89, 332
INCLUDE pseudo-op, 153
  macros and, 173
index register, 59
initialization of programs, 195
instruction pointer, 45, 49, 58
INT instruction, 122, 333
interpretive languages, 18
interrupts, 63, 74, 121-124. See also name of interrupt
interrupt controller, 301
interrupt enable register, 298
interrupt flag, 62
interrupt identification register, 298
interrupt service routine, 307, 309
interrupt vectors, 123
INTO instruction, 124, 333
INTR (external mask interrupt) instruction, 333
I/O (input/output), 50
  disk, 204-247
IRET (interrupt return) instruction, 50, 124, 333
IRP pseudo-op, 175, 177
IRPC pseudo-op, 175, 177
iteration control instructions, 74, 119-121

JCXZ instruction, 121, 334
JMP instruction, 30, 31, 114
Jump if... instructions, 116, 333-336

keyboard buffer, programming and, 232
KEYDSP program, 399

label field, 29
labels for numerical values, 23-26, 142
  local, 173
  syntax for, 29
LABEL pseudo-op, 146
LAHF (load register AH from flags) instruction, 86, 336
.LALL pseudo-op, 159, 160
  languages, types of, 17-21
LDS (load pointer using DS) instruction, 85, 336
LE operator, 166
LEA (load effective address) instruction, 84, 336
LENGTH operator, 144
LES (load pointer using ES) instruction, 85, 336
.LFCOND pseudo-op, 159, 161
line control register, 296
line drawing, 260
line status register, 299
linkers, 27
  use of, 38

SCAS (Scan String) instructions, 128, 342
scroll function in BIOS, 233
SDLC protocol, 284
segment attribute for labels, 142
segment override operators, 165, 342
segment register, 56
SEGMET pseudo-op, 138
segmented memory, 58
  programming and, 193
semi-colon (;), 176
sequential file access, 221
serial I/O functions, 251
serial board port assignments, 273, 295
serial data transmission, 271
SF (sign flag), 62
.SFCOND pseudo-op, 159, 161
shift instructions, 73, 107-110
SHL (Shift Logical Left) instruction, 109, 342
SHL operator, 167
SHORT operator, 165
SHR (Logical Shift Right) instruction, 109
SHR operator, 167
sign extensions, 97
sign flag, 62
simplex, 269, 281
SINGLE STEP interrupt, 343
  SIZE operator, 144
slash (/) operator (division), 167
sorting, 244
sound generation, 263
SOUND.ASM program, 440
source code
  creation of, 28-32
  defined, xii
  flow charting and, 34-47
SP (stack pointer) register, 59
spooling, 315
SS operator, 165
stacks, 48
stack pointer, 45, 48
stack pointer register, 59
stack segment, programming and, 192, 233
status flags, 60
status register, 45
STC (set carry) instruction, 131, 343
STD (set direction flag) instruction, 343
STI (Set Interrupt flag)instruction, 132, 343
STOS (Store String) instructions, 130, 343
string instructions, 74, 124-130
STRUC pseudo-op, 152
SUB (subtraction) instruction, 91, 344
subroutines, 48
subtraction, binary, 5
subtraction instructions, 73, 91-95
SUBTTL pseudo-op, 159
symbolic representation for numerical values, 23-26
SYN control code, 283
synchronous communications, 283
syntax of source statement, 28-32

TANDY 2000 MS-DOS, 250
telecommunications, 272-276
  frequency assignments for, 276
TEST instruction, 106, 344
.TFCOND pseudo-op, 159, 161
THIS operator, 164
THR (transmitter holding register), 297
TITLE pseudo-op, 159, 161
TR (trap flag), 62
translate (XLAT) instruction, 82, 344
transmitting characters function, 303
  via BIOS, 304
trap flag, 62
tree structures, 227
type attribute for labels, 143
type zero interrupt, 99

UART (Universal Asynchronous Receiver Transmitter), 272
unconditional transfer instructions, 74, 112-115

video I/O, 253

WAIT instruction, 132, 344
word, defined, 2
WR signal, 51
writing blocks, 225
writing files, 222
writing records, 224

.XALL pseudo-op, 159, 160
XCHG (exchange) instruction, 82
XLAT (translate) instruction, 82
.XLIST pseudo-op, 159
XON/XOFF, 12
XOR (exclusive OR) instruction, 105, 345
XOR operator, 167
.XREF pseudo-op, 159, 160

ZF (zero flag), 46, 61

.LIST pseudo-op, 159
listing file, 39
listing pseudo-ops, 158
local labels, 173
LOCAL pseudo-op, 173
LOCK prefix, 133, 337
LODS (Load String) instructions, 129, 337
logic instructions, 73, 102-107
logical disk sectors, 230
logical operators, 167
Logical Shift Right (SHR) instruction, 109, 343
LOOP instructions, 119, 127, 337
LOW operator, 164
LSB (least significant bit), defined, 3
LT operator, 166

MACFLE.MAC program, 369
machine code, xiii
machine language, 19
macro definitions, 169
MACRO pseudo-op, 175
maskable interrupts, 64
maximum mode control signals, 51
medium resolution graphics mode, 256
memory, 46
  addressing of, 56
  map of, 55, 57
  segmentation of, 47, 56
memory direct addressing, 65
  with index, 66
memory refresh, 46
messages, programming and, 193, 233
microprocessors, 17
minimum mode control signals, 51
minus sign (-) operator, 167
M/IO signal, 51
mnemonics, 22, 30
MOD operator, 167
modems, 288-292
modem control register, 297
modem status register, 300
monochrome character attributes, 253
MOV instruction, 17, 20, 31, 76, 338
MOVS (move string) instructions, 126, 338
MS-DOS, 181-188
  directory structure of, 214
  disk functions in, 207
  disk map of, 212
  file access in, 221-229
  programming tools in, 213-221
  RS-232C functions in, 302
  serial I/O in, 293
MS-DOS function, 219
MSB (most significant bit), defined, 3
MUL instruction, 96, 338
multiplication indicators, 62
multiplication instructions, 73, 95-99

NAK (negative acnowledgement), 284
NAME pseudo-op, 153
names, symbolic, 142
NE operator, 166
NEAR procedure, defined, 113
NEG (negate) instruction, 94, 33
NEW—COPY program, 416
NEW—TYPE.ASM program, 412
NEXT statement, 119
nibble, defined, 8
NMI interrupt, 64, 337
no operation instruction, 75, 131, 339
noise on line, 285
nonmaskable interrupt, 64
NOT instruction, 107, 339
NOT operator, 167
null cable, 291
number conversions, program for, 189-205
numbering systems, 2-13
  program fr conversions of, 189-205
NUMBERSY.ASM program, 401

object file, 39
OF (overflow flag), 62
OFFSET attribute for labels, 143
op-codes, 30
operands, 31
  destination, 71
operating systems, 181-188
operators, 162-168. See also name of operator
OR instruction, 105, 339
OR operator, 167
ORG pseudo-op, 148
OUT instruction, 83, 339
%OUT pseudo-op, 159
overflow flag, 62
overrun error, 300

PAGE pseudo-op, 19, 161
parallel data transmission, 270
parity, 287
parity error, 300
parity flag, 61
parsing a string, 235
pathnames, 227
percentage sign (%), 176, 180
PF (parity flag), 61
phase modulation, 277
physicaldisk sectors, 230
pipelining, 44
pixel positioning, 257
plus sign (+) operator, 167
pointer register, 59
pointers, 45
POP instruction, 50, 78, 339
POPF instruction, 86, 340
port addresses, 25
  IN and OUT instruction and, 83
port assignments, serial board, 273, 295
power bus, 51
printer I/O, 252
PROC pseudo-op, 139
procedures, 48
processor control instructions, 75, 130-134
program segment prefix (PSP), 215-217
program transfer instructions, 74, 112-124
programmer, 16
programming techniques, 189-205
  BIOS and, 259-263
  disk I/O and, 207-247
protocols for communications, 268
pseudo-ops, 136-140, 145-162. See also name of pseudo-op
PTR (pointer) operator, 162
PUBLIC pseudo-op, 147
PURGE pseudo-op, 175, 176
PUSH instruction, 50, 78, 340
PUSHF (push flags) instruction, 86, 340

.RADIX pseudo-op, 154
RAM (Random Access Memory), 46
  communications and, 118
random block access, 224
random record access, 223
RCL (Rotate through Carry Left) instruction, 111, 340
RCR (Rotate through Carry Right) instruction, 111, 340PRD signal, 51
reading blocks, 225
reading files, 222
reading records, 223, 236
receive function, 302
  via BIOS, 304
RECORD pseudo-op, 150
registers, 43, 44. See also name of register
  specialized, 45
register direct addressing, 65
register indirect addressing, 66
  with displacement addressing, 67
relational operators, 166
REP instructions, 125, 340-341
REPT pseudo-op, 175, 176
RET instruction, 49, 112-114, 31
  POP instructions and, 79
RHR (receiver holding register), 297
ROL (Rotate Left) instruction, 110, 341
ROM (Read Only Memory), 46
ROR (Rotate Right) instruction, 110, 341
rotate instructions, 73, 110
RS-232C, 251, 289
  signal and pin assignments for, 290, 291

SAHF (store register AH into flags) instruction, 86, 341
SAL (Shift Arithmetic Left) instruction, 107-109, 342
.SALL pseudo-op, 159, 160
SAR (Shift Arithmetic Right) instruction, 107, 342
SBB (subtract with borrow) instruction, 92, 342

Source Code Offer

Gary A. Shade has made available to interested readers the complete source code for every programming example used in his book *8088 IBM PC Assembly Language Programming*.

Why spend hours entering the source examples found in this book when you can order the source code on IBM PC compatible media? The entire source code for the macro library found in Chapter 6 is worth the price of the diskette alone. You can use the macro library in each and every one of your own programs—use routines from any of the examples included in your own programs with no royalties to pay!

To order:

Include $29.95 money order, certified check, or personal check (*). Mail your check (made payable to Argonaut Systems) and the form below to:

Argonaut Systems
POB 2492
Northbrook, IL 60065

Send me the complete source library to "8088 IBM PC Assembly Language Programming" by Gary A. Shade. I have enclosed a certified check, money order or personal check for $29.95 made payable to Argonaut Systems.

Name ____

Address ____

City ____

State ___________________________ Zip _______________

* Allow 3-4 weeks for delivery. Personal checks - allow 4-5 weeks for delivery.

FPT >$17.95

8088
IBM PC
Assembly Language Programming

Gary Shade

The Nuts 'N Bolts of Assembly Language Programming

Professionals and hobbyists! Whether you're a serious computer user or just love to tinker with your PC, this guide shows you the best ways to attack Assembly Language programming.

You'll get a step-by-step introduction to microprocessors . . . numbering systems . . . and character encoding to give you the boost you need to get the most out of ALP.

And that's not all! The book covers MS DOS function calls and features number conversion charts . . . ASCII and BAUDOT character encoding charts . . . and a complete set of original Intel Corporation data sheets!

ISBN 0-03-001298-8
Ret:0785:001745: